Processes

Daniel Hagimont (INPT)

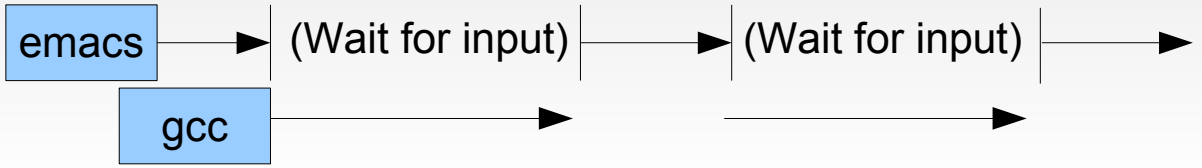
Thanks to Noël De Palma, Fabienne Boyer and Arnaud Legrand (UJF)

Process

- A process is an instance of a running program
  - Eg : gcc, sh, firefox ...
  - Created by the system or by an application
  - Created by a parent process
  - Uniquely idendified (PID)
- Correspond to two units :
  - Execution unit
    - Sequential control flow (execute a flow of instructions)
  - Addressing unit
    - Each process has its own address space
    - Isolation

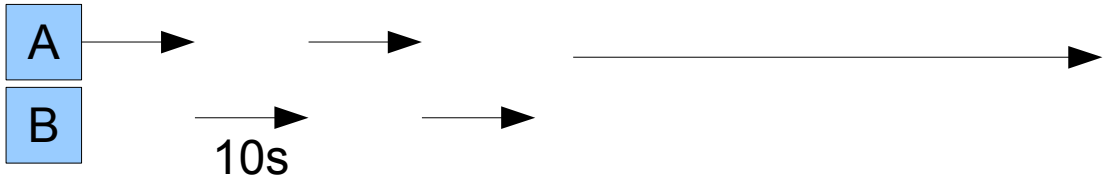
Concurent processes

- Multiple processes can increase CPU utilization
  - Overlap one process's computation with another's wait



- Multiple processes can reduce latency
  - Running A then B requires 100 secs for B to complete

  - Running A and B concurently improves the average response time



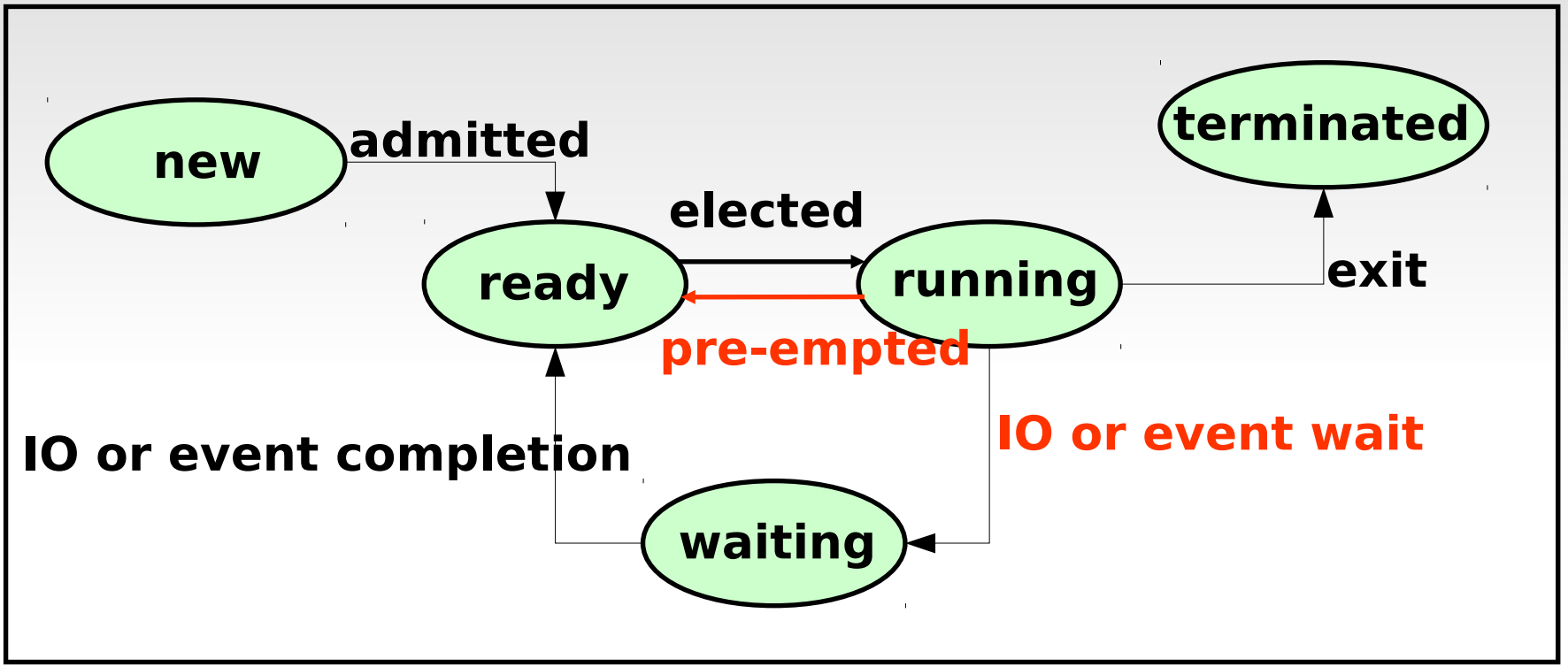
Execution context

- A process is characterized by its context
- Process' current state
  - Memory image
    - Code of the running program
    - Static and dynamic data
  - Register's state
    - Program counter (PC), Stack pointer (SP) ...
  - List of open files
  - Environment Variables
  - …
- To be saved when the process is switched off
- To be restored when the process is switched on

Running mode

- User mode
  - Access restricted to process own adress space
  - Limited instruction set
- Supervisor mode
  - Full memory access
  - Full access to the instruction set
- Interrupt, trap
  - Asynchronous event
  - Illegal instruction
  - System call request

Process Lifecycle



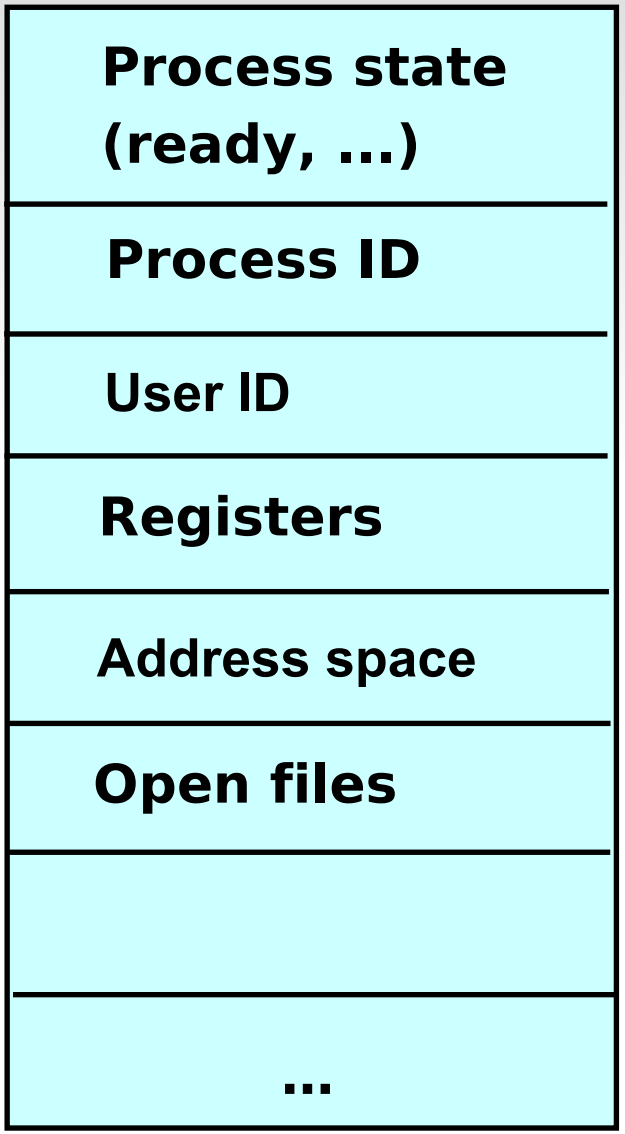
- Which process should the kernel run ?
  - If 0 runnable, run a watchdog, if 1 runnable, run it
  - If n runnable, make scheduling decision

Process management by the OS

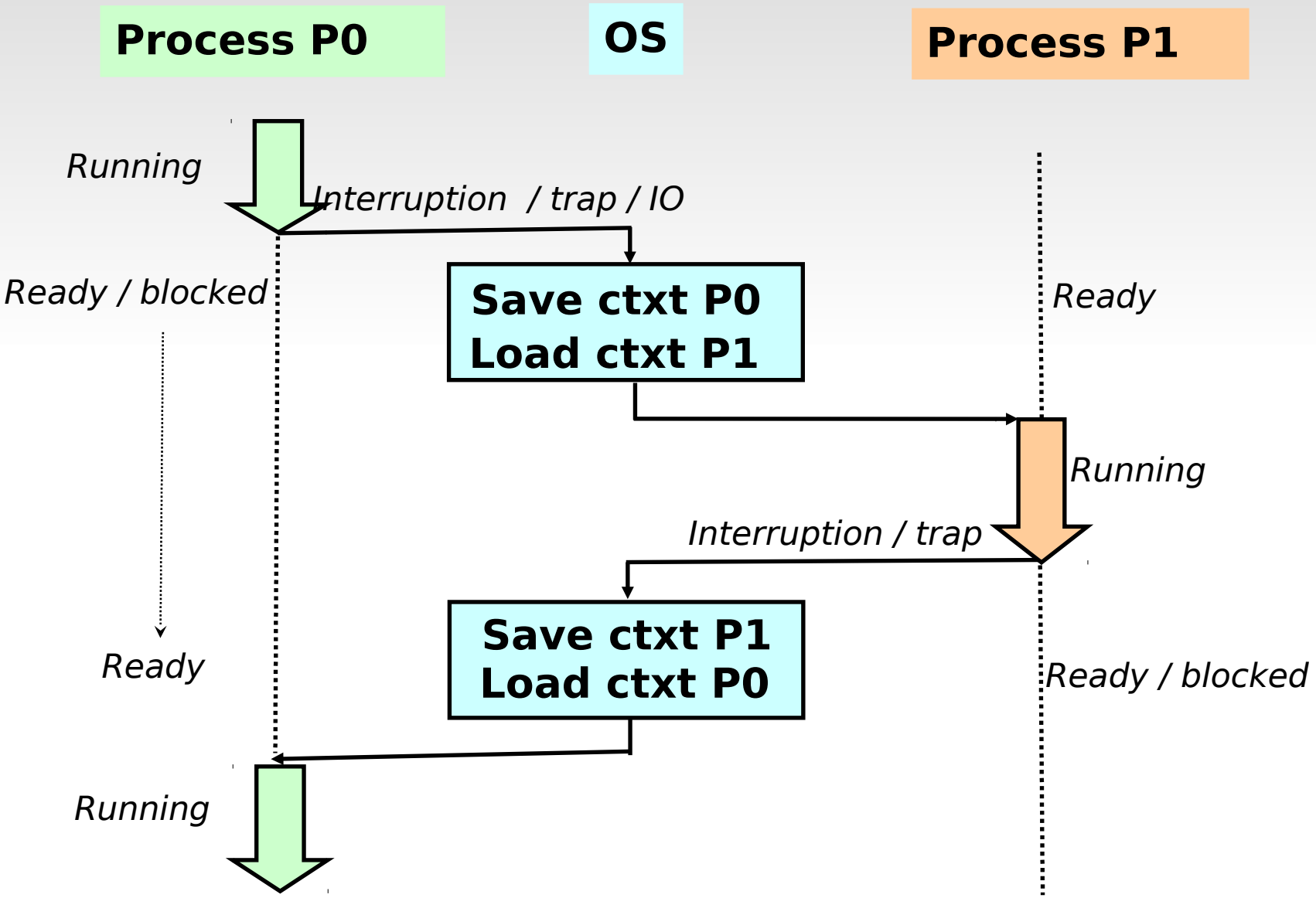
- Process queues
  - Ready queue (ready processes)
  - Device queue (Process waiting for IO)
  - Blocked Queue (Process waiting for an event)
  - ...
- OS migrates processes across queues

Process Control Structure

- Hold a process execution context
- PCB (Process Control Block):
  - Data required by the OS to manage process
- Process tables:
  - PCB [MAX-PROCESSES]



Process context switch



CPU Allocation to processes

- The scheduler is the OS's part that manages CPU allocation
- Criteria / Scheduling Algorithm
  - Fair (no starvation)
  - Minimize the waiting time for processes
  - Maximize the efficiency (number of jobs per unit of time)

Simple scheduling algorithms (1/2)

- Non-pre-emptive scheduler
  - FCFS (First Come First Served)
    - Fair, maximize efficiency
- Pre-emptive scheduler
  - SJF (Shortest Job First)
    - Priority to shortest task
    - Require to know the execution time (model estimated from previous execution)
    - Unfair but optimal in term of response time
  - Round Robin (fixed quantum)
    - Each processus is affected a CPU quantum (10-100 ms) before pre-emption
    - Efficient (unless the quantum is too small), fair / response time (unless the quantum too long)

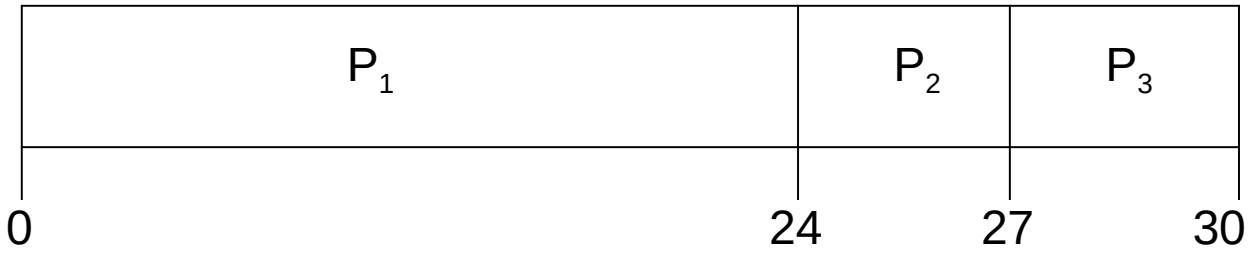
Simple scheduling algorithms (2/2)

- Round robin with dynamic priority
  - A priority associated with each process
  - One ready queue per priority level
  - Decrease priority for long tasks (prevent starvation)
- Round robin with variable quantum
  - N^{th} scheduling receives 2^{N-1} quantum (reduce context-switches)

First-Come, First-Served (FCFS) non pre-emptive (1/2)

Process's	execution time
P1	24
P2	3
P3	3

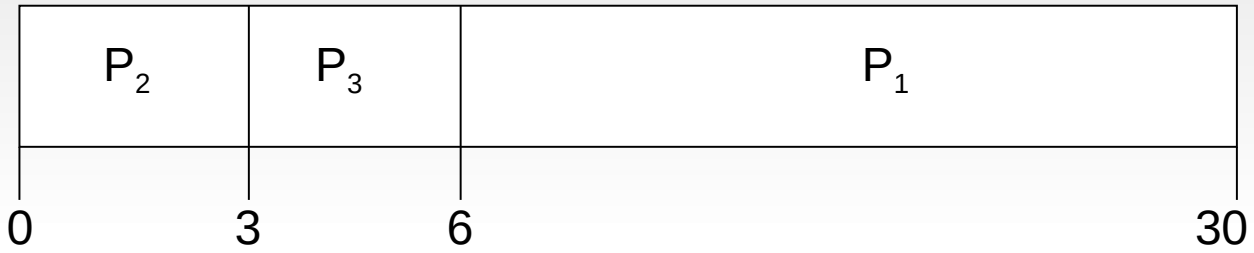
- Let's these processes come in this order : P1,P2,P3



- Response time of P1 = 24; P2 = 27; P3 = 30
- Mean time : (24 + 27 + 30)/3 = 27

First-Come, First-Served (FCFS) (2/2)

- Let's these processes come in this order : P2 , P3 , P1 .



- Response time : P1 = 30; P2 = 3; P3 = 6
- Mean time : (30 + 3 + 6)/3 = 13
- Better than the precedent case
- Schedule short processes before

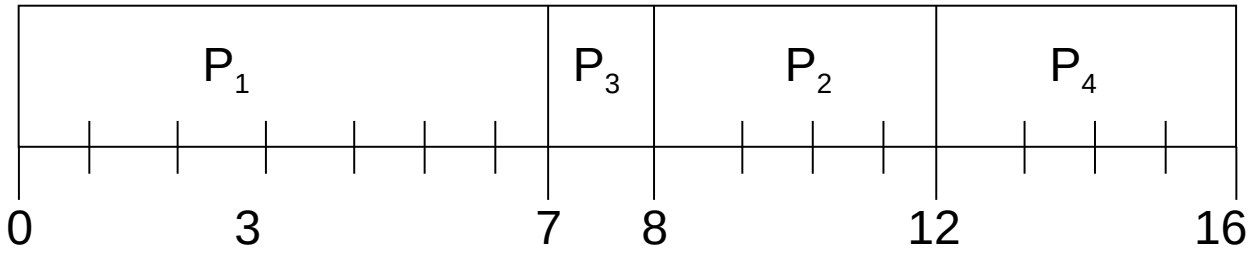
Shortest-Job-First (SJF)

- Associate with each process its execution time
- Two possibilities :
  - Non pre-emptive – When a CPU is allocated to a process, it cannot be pre-empted
  - Pre-emptive – if a new process comes with a shorter execution time than the running one, this last process is pre-empted (Shortest-Remaining-Time-First - SRTF)
- SJF is optimal / mean response time

Non Pre-emptive SJF

Process	Come in	Exec. Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (non pre-emptive)

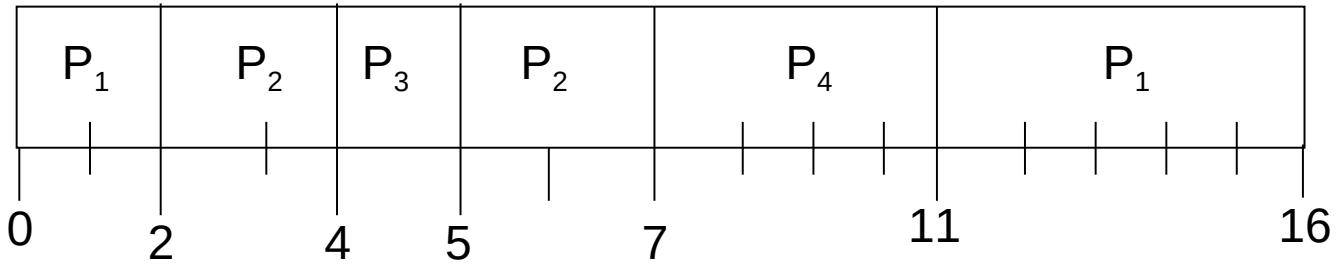


- Mean response time = (7 + 8 + 12 + 16)/4 = 10,75

Pre-emptive SJF (SRTF)

Process	Come in	Exec. Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (pre-emptive)

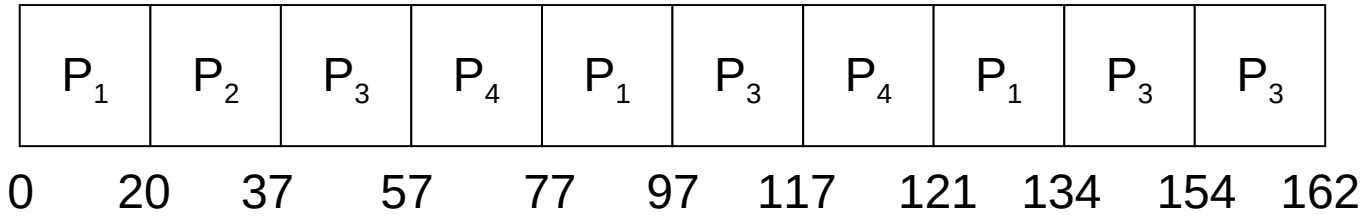


- Mean response time = (16 + 7 + 5 + 11)/4 = 8,25

Round Robin (Quantum = 20ms)

Process	Exec Time
P1	53
P2	17
P3	68
P4	24

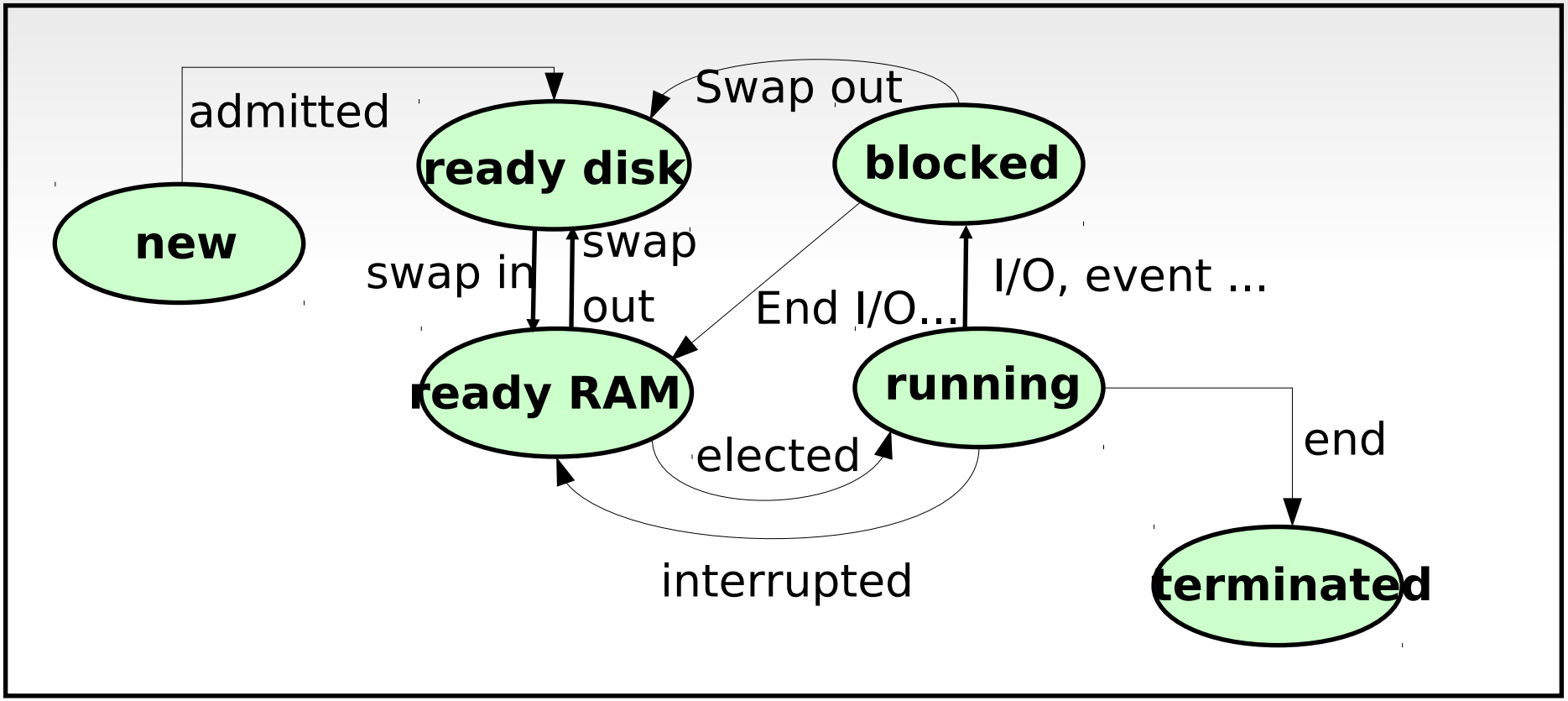
- Efficiency and mean response worse than SJF



Multiple level scheduling algorithm

- The set of ready processes too big to fit in memory
- Part of these processes are swapped out to disk. This increases their activation time
- The elected process is always choosen from those that are in memory
- In parallel, another scheduling algorithm is used to manage the migration of ready process between disk and memory

Two level scheduling



Process SVC overview

- int fork (void);
  - Creates a new process that is an exact copy of the current one
  - Returns the process ID of the new process in the “parent”
  - Returns 0 in “child”
- int waitpid (int pid, ...);
  - pid – the process to wait for, or -1 for any
  - Returns pid of resuming process or -1 on error
- Hierarchy of processes
  - run the pstree -p command

Process SVC overview

- void exit (int status);
  - Current process stops
  - status: returned to waitpid (shifted)
  - By convention, status of 0 is success
- int kill (int pid, int sig);
  - Sends signal sig to process pid
  - SIGTERM most common value, kills process by default (but application can catch it for “cleanup”)
  - SIGKILL stronger, kills process always
- When a parent process terminates before its child, 2 options:
  - Cascading termination (VMS)
  - Re-parent the orphan (UNIX)

Process SVC overview

- int execve (const char *prog, const char **argv, char **envp;)
  - prog – full pathname of program to run
  - argv – argument vector that gets passed to main
  - envp – environment variables, e.g., PATH, HOME
- Generally called through a wrapper functions
  - int execvp (char *prog, char **argv);
    - Search PATH for prog, use current environment

Fork and Exec

- The fork system call creates a copy of the PCB
  - Opened files and memory mapped files are thus similar
  - Open files are thus opened by both father and child. They should both close the files.
  - The pages of many read only memory segments are shared (text, r/o data)
  - Many others are lazily copied (copy on write)
- The exec system call replaces the address space, the registers, the program counter by those of the program to exec.
  - But opened files are inherited

Why fork

- Most calls to fork followed by execvp
- Real win is simplicity of interface
  - Tons of things you might want to do to child
  - Fork requires no arguments at all
  - Without fork, require tons of different options
  - Example: Windows CreateProcess system call

```
Bool CreateProcess(
    LPCTSTR lpApplicationName, //pointer to a name to executable module
    LPTSTR lpCommandLine, // pointer to a command line string
    LPSECURITYATTRIBUTES lpProcessAttributes, //process security attr
    LPSECURITYATTRIBUTES lpThreadAttributes, // thread security attr
    BOOL bInheritHandles, //creation flag
    DWORD dwCreationFlags, // creation flags
    LPVOID lpEnvironnement, // pointer to  new environment block
    LPCTSTR lpCurrentDirectory, // pointer to crrent directory name
    LPSTARTUPINFO lpStartupInfo, //pointer to STARTUPINFO
    LPPROCESSINFORMATION lpProcessInformaton // pointer to PROCESSINFORMATION );
```

Fork example

- Process creation
  - Done by cloning an existing process
    - Duplicate the process
  - Fork() system call
    - Return 0 to the child process
    - Return the child's pid to the father
    - Return -1 if error

```
#include <unistd.h>

pid_t fork(void);
```

```
r = fork();
if (r==-1) ...    /* error */
else if (r==0) ... /* child's code */
else ...          /* father's code */
```

Fork example

- How many processes are created ?

```
fork();
fork();
fork();
```

```
for (i=0; i<3;i++){
   fork();
}
```

- What are the possible different traces ?

```
int i = 0;
switch((j=fork()) {
        case -1 : perror("fork"); break;
        case  0 : i++; printf("child I :%d",i); break;
        default : printf("father I :%d",i);

}
```

Exec example

- Reminder: main function definition
  - int main(int argc, char *argv[]);
- Execvp call
  - Replaces the process's memory image
  - int execvp(const char *file, const char *argv[]);
    - file : file name to load
    - argv : process parameters
  - execvp calls main(argc, argv) in the process to launch

Exec example

```
char  * argv[3];


argv[0] = “ls “;
argv[1] = “-al “;
argv[2] = 0;
execvp(“ls“, argv);
```

Father/child synchronization

- The father process waits for the completion of one of its child
  - pid_t wait(int *status):
    - The father waits for the completion of one of its child
      - pid_t : dead child's pid or -1 if no child
      - status : information on the child's death
  - pid_t waitpid(pid_t pid, int *status, int option);
    - Wait for a specific child's death
    - Option : non blocking … see man

Wait example

```
#include <sys/types.h>
#include <sys/wait.h>

main(){
  int spid, status;
  switch(spid = fork()){
            case -1 : perror(...); exit(-1);
            case 0  :  // child's code
                       break;
            default : // the father wait for this child's terminaison
                       if (waitpid(spid,&status,0)==-1) {perror(...);exit(-1);}
                       ...
  }
}
```

Example: minishell

```
#include <stdio.h>
#include <stdlib.h>

pid_t pid;
char *av[2];
char cmd[20];

void doexec() {
if (execvp(av[0],av)==-1)
            perror ("execvp failed");
    exit(0);
}
```

```
int main() {
for (;;) {
    printf(">");
    scanf("%s",cmd);
    av[0] = cmd;
    av[1] = NULL;
    switch (pid = fork()) {
        case -1: perror("fork"); break;
        case 0:
            doexec();
        default:
            if (waitpid(pid, NULL, 0) == -1)
                perror ("waitpid failed");
   }
}
}
```

I/O redirection

- A file is addressed through a descriptor
  - 0, 1 et 2 correspond to standard input, standard output, and standard error
  - The file descriptor number is returned by the open system call
- Basic operation
  - int open(const char *pathname, int flags);
    - O_RDONLY, O_WRONLY, O_RDWR …
  - int creat(const char *pathname, mode_t mode);
  - int close(int fd)
  - ssize_t read(int fd, void *buf, size_t count);
  - ssize_t write(int fd, void *buf, size_t count);

I/O redirection

- Descriptor duplication
  - dup(int oldfd); dup2(int oldfd, int newfd);
  - Used to redirect standard I/O

```
#include <stdio.h>
#include <unistd.h>
int f;
/* redirect std input */

...
close(STDIN_FILENO); // close std input
dup(f);              // dupliquate f on the first free descriptor (i.e. 0)
close(f);            // free f
...
```

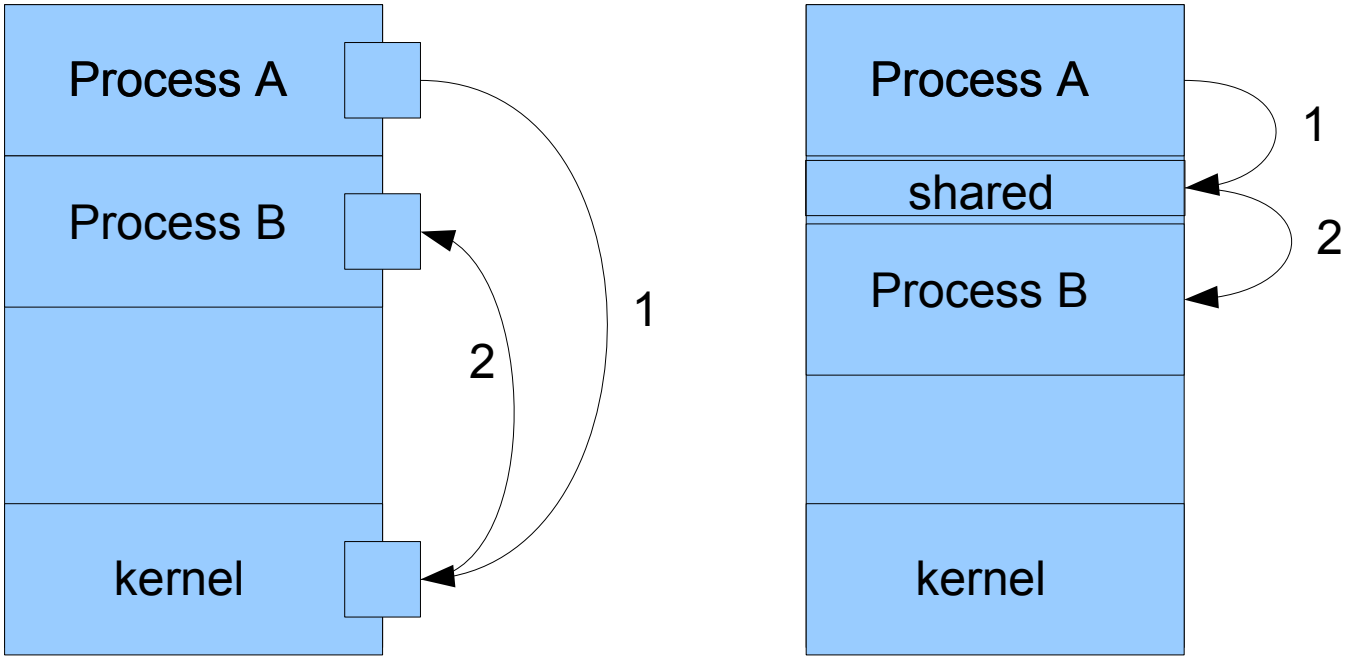
```
dup2(f,STDIN_FILENO);
close(f);
```

Cooperation between processes

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process. Advantages:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

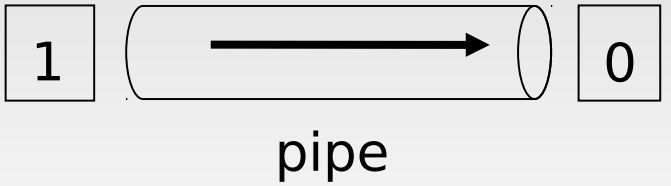
Process Interaction

- How can processes interact in real time?
  - Through files but it's not really "real time".
  - Through asynchronous signals or alerts
  - By sharing a region of physical memory
  - By passing messages through the kernel/network



Pipe

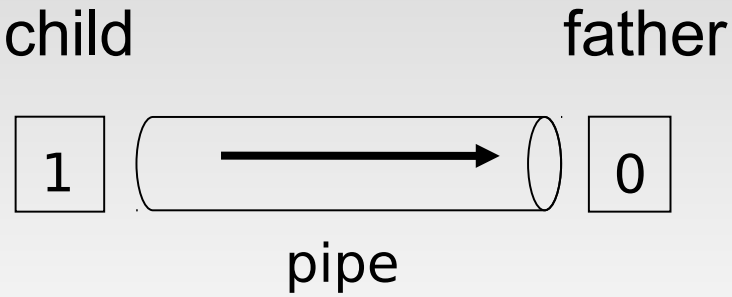
- Communication mechanism between processes
  - Fifo structure
  - Limited capacity
  - Producer/consumer synchronization



- int pipe (int fds[2]);
  - Returns two file descriptors in fds[0] and fds[1]
  - Writes to fds[1] will be read on fds[0]
  - Returns 0 on success, -1 on error
- Operations on pipes
  - read/write/close – as with files
  - When fds[1] closed, read(fds[0]) returns 0 bytes (EOF)
  - When fds[0] closed, write(fds[1]): kill process with SIGPIPE

Pipe example

```
void doexec (void) {
    int pipefds[2];
     pipe (pipefds);
        switch (fork ()) {
            case -1:        perror ("fork"); exit (1);
            case 0:
                        dup2 (pipefds[1], 1);
                        close (pipefds[0]); close (pipefds[1]);
                        execvp(...);
                        break;
           default:
                        dup2 (pipefds[0], 0);
                        close (pipefds[0]); close (pipefds[1]);
                        break;
        }
/* ... */
}
```



Asynchronous notification (Signal)

- A process may send a SIGSTOP, SIGTERM, SIGKILL signal to suspend (CTRL-Z), terminate or kill a process using the kill function:
  - int kill (int pid, int sig);
  - A lot of signals … see man pages
  - Some signals cannot be blocked (SIGSTOP and SIGKILL)
- Upon reception of a signal, a given handler is called. This handler can be obtained and modified using the signal function:
  - typedef void (*sighandler_t)(int); // handler
  - sighandler_t signal(int signum, sighandler_t handler); // set a handler

Signal example

```
void handler(int signal_num) {
    printf("Signal %d => ", signal_num);
    switch (signal_num) {
    case SIGTSTP:
        printf("pause\n");
        break;
    case SIGINT:
    case SIGTERM:
        printf("End of the program\n");
        exit(0);
        break;
    }
}
```

```
int main(void) {
  signal(SIGTSTP, handler);
  /* if control-Z */
  signal(SIGINT, handler);
  /* if control-C */
  signal(SIGTERM, handler);
  /* if kill process */
  while (1) {
    sleep(1);
    printf(".\n");
   }
printf("end");
exit(0);
}
```

- Signal handling is vulnerable to race conditions: another signal (even of the same type) can be delivered to the process during execution of the signal handling routine.
- The sigprocmask() call can be used to block and unblock delivery of signals.

Shared memory segment

- A process can create/use a shared memory segment using:
  - int shmget(key_t key, size_t size, int shmflg);
  - The returned value identifies the segment and is called the shmid
  - The key is used so that process indeed get the same segment.
- The owner of a shared memory segment can control access rights with shmctl()
- Once created, a shared segment should be attached to a process address space using
  - void *shmat(int shmid, const void *shmaddr, int shmflg);
- It can be detached using int shmdt(const void *shmaddr);
- Can also be done with the mmap function
- Example

Shared memory example

```
int shmid;
void *shm;
key_t key = 1234;
/* Create the segment */
if ((shmid = shmget(key, 10,
            IPC_CREAT | 0666)) < 0) {
            perror("shmget failed");
            exit(1);
}
/* Attach the segment */
if ((shm = shmat(shmid, NULL, 0)) ==
                        (void *) -1) {
           perror("shmat failed");
           exit(1);
}
```

```
int shmid;
void *shm;
key_t key = 1234;
/* Get the segment */
if ((shmid = shmget(key, 10, 0666))< 0) {
           perror("shmget failed");
           exit(1);
}
/* Attach the segment */
if ((shm = shmat(shmid, NULL, 0)) == (void *) -1) {
           perror("shmat failed");
           exit(1);
}
```

Message queue

- Creation of a message queue
  - int msgget(key_t key, int msgflg);
- Control of the message queue
  - int msgctl(int msqid, int cmd, struct msqid_ds *buf);
- Emission of a message
  - int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
- Reception of a message
  - int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
- Example

Message queue example

```
int msgid;
key_t key = 1234;
char buffer[1024];
struct msgbuff msg;
/* Create the queue */
if ((msgid = msgget(key,
        IPC_CREAT | 0666)) < 0) {
        perror("msgget failedt");
        exit(1);
}
/* send a message */
msg.mtype=1;
strcpy(msg.mtext, buffer);
if ((msgsnd(msgid, (void *)&msg,
            1024,0)) == -1) {
            perror("msgsnd failed");
            exit(1);
}
```

```
int msgid;
key_t key = 1234;
char buffer[1024];
struct msgbuff msg;
/* get the queue */
if ((msgid = msgget(key, 0666)) < 0) {
        perror("msgget failedt");
        exit(1);
}
/* receive a message */
if ((msgrcv(msgid, (void *)&msg,
        1024,0,0)) == -1) {
        perror("msgsnd failed");
        exit(1);
}
strcpy(buffer, msg.mtext);
```

Socket

- A socket is defined as an endpoint for communication
- Used for remote communication
- Basic message passing API
- Identified by an IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication between a pair of sockets and bidirectionnal
- => second part of Teaching Unit (networking)

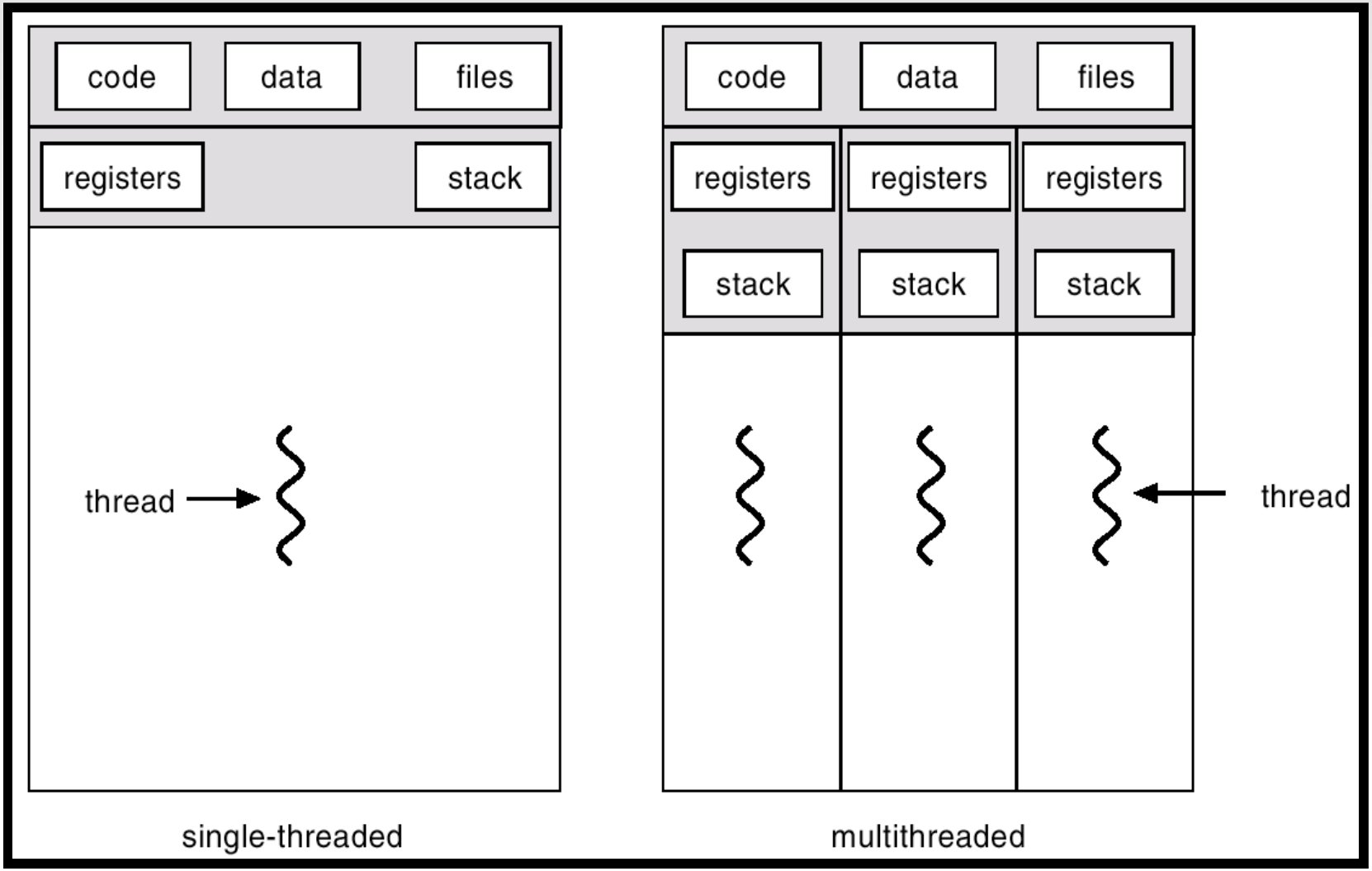
Process

- Unix process: heavy
  - Context: large data structure (includes an address space)
  - Protected address space
    - Address space not accessible from another process
    - Sharing / communication
      - At creation time (fork)
      - Via shared memory segments
      - Via messages (queues, sockets)
  - Communication is costly

Threads

- Light weight process
  - Light weight context
    - A shared context: address space, open files …
    - A private context: stack, registers …
- Faster communication within the same address space
  - Message exchange, shared memeory, synchronization
- Useful for concurrent/parallel applications
  - Easier
  - More efficient
  - Multi-core processors

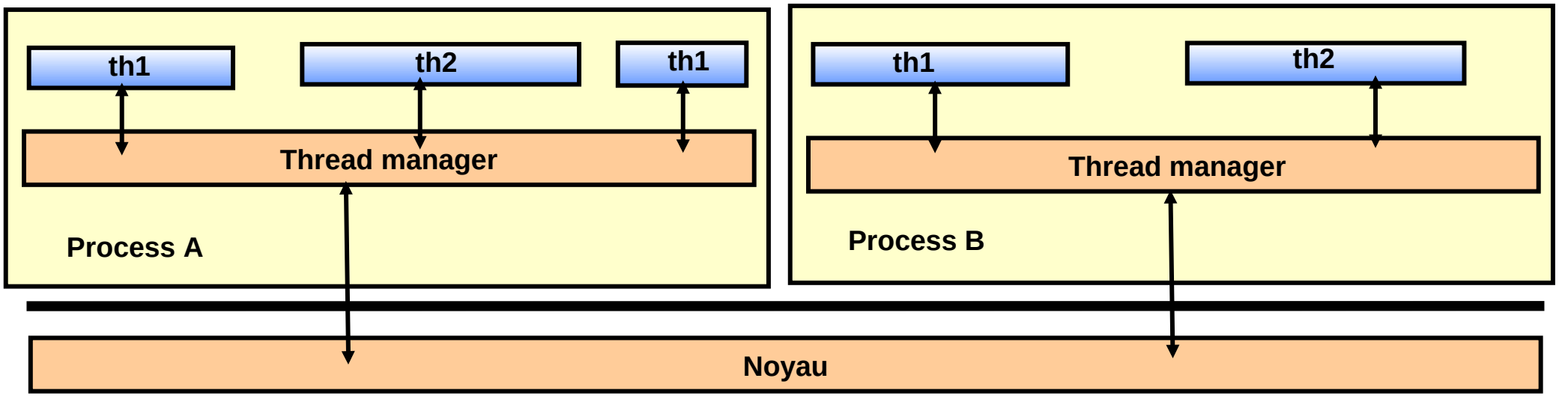
Single-threaded vs multi-threaded processes



A.Sylberschatz

User-level Threads

- Implemented in a user level library
- Unmodified Kernel
- Threads and thread scheduler run in the same user process
  - Examples: POSIX Pthreads, Mach C-threads, Solaris threads

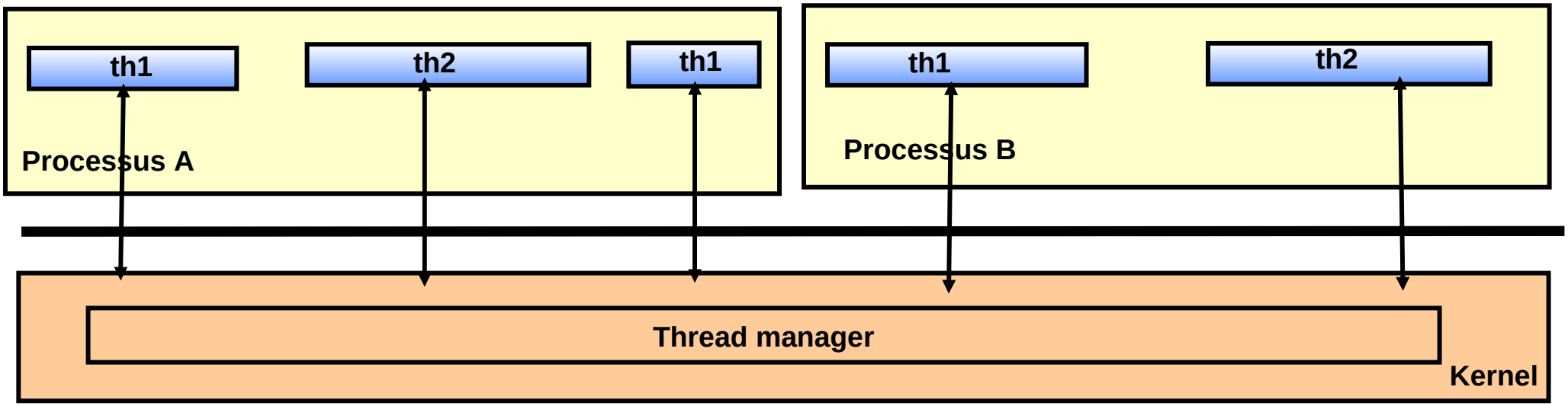


Advantages and disadvantages of User-level threads

- Parallelism (-)
  - No real parallelism between the threads within a process
- Efficiency (+)
  - Quick context switch
- Blocking system call (-)
  - The process is blocked in the kernel
  - All thread are blocked until the system call (I/O) is terminated

Kernel level threads

- Thread managed by the kernel
- Thread creation as a system call
- When a thread is blocked, the processor is allocated to another thread by the kernel
  - Examples: Windows NT/2000, Solaris, Tru64 UNIX, Linux



Advantages and disadvantages of Kernel-level threads

- Blocking system call (+)
  - When a thread is blocked due to an SVC call, threads in the same process are not
- Real Parallelism (+)
  - N threads in the same process can run on K processors (multi-core)
- Efficiency (-)
  - More expensive context switch / user level threads
  - Every management operation goes throught the kernel
  - Require more memory

POSIX Threads : Pthreads API

- int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);
  - Creates a thread
- pthread_t pthread_self (void);
  - Returns id of the current thread
- int pthread_equal (pthread_t thr1, pthread_t thr2);
  - Compare 2 thread ids
- void pthread_exit (void *status);
  - Terminates the current thread
- int pthread_join (pthread_t thr, void **status);
  - Waits for completion of a thread
- int pthread_yield(void);
  - Relinquish the processor
- Plus lots of support for synchronization [next lecture]

Pthread example (1/2)

```
#include <pthread.h>

void * ALL_IS_OK = (void *)123456789L;

char *mess[2] = { "thread1", "thread2" };

void * writer(void * arg)
{
   int i, j;

   for(i=0;i<10;i++) {
      printf("Hi %s! (I'm %lx)\n", (char *) arg, pthread_self());
      j = 800000; while(j!=0) j--;
   }

   return ALL_IS_OK;
}
```

Pthread example (2/2)

```
int main(void)
{ void * status;
  pthread_t writer1_pid, writer2_pid;

   pthread_create(&writer1_pid, NULL, writer, (void *)mess[1]);
   pthread_create(&writer2_pid, NULL, writer, (void *)mess[0]);

   pthread_join(writer1_pid, &status);
   if(status == ALL_IS_OK)
      printf("Thread %lx completed ok.\n", writer1_pid);

   pthread_join(writer2_pid, &status);
   if(status == ALL_IS_OK)
      printf("Thread %lx completed ok.\n", writer2_pid);

   return 0;
}
```

Fork(), exec()

- What happens if one thread of a program calls fork()?
  - Does the new process duplicate all threads ? Or is the newprocess single-threaded ?
  - Some UNIX systems have chosen to have two versions of fork()
- What happens if one thread of a program calls exec()?
  - Generally, the new program replace the entire process, including all threads.

Resources you can read

- Pthreads
  - https://computing.llnl.gov/tutorials/pthreads/
- Operating System Concepts, 8th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 3, 4 & 5
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.1 & 2.2)