

# Chapter 3: Processes- Concept

Prof. Li-Pin Chang  
National Chiao Tung University

# Chapter 3: Processes-Concept

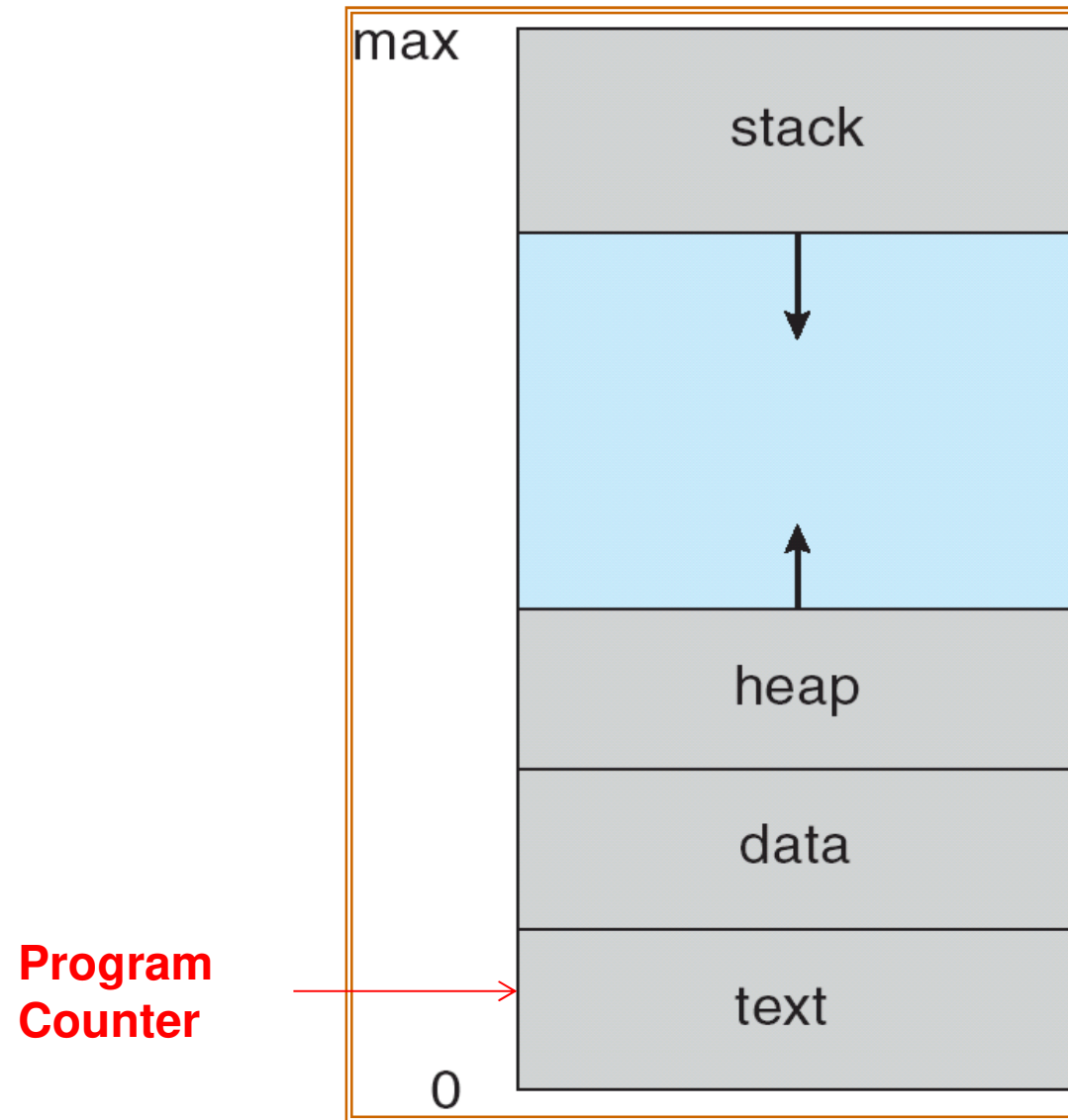
- Process Concepts
- Process Schedulers
- Operations on Processes
- Inter-process Communication
- Examples of IPC Systems

# PROCESS CONCEPTS

# Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
  - Textbook uses the terms **job**, **task**, and **process** almost interchangeably
- **Process – a program in execution**; process execution must progress in sequential fashion
  - Process: active, program: passive
- A process includes:
  - Text section
  - Stack section
  - Data section
    - BSS and variables with initial values
  - Heap
  - Program counter and other CPU registers

# Process in Memory (virtual addr space)

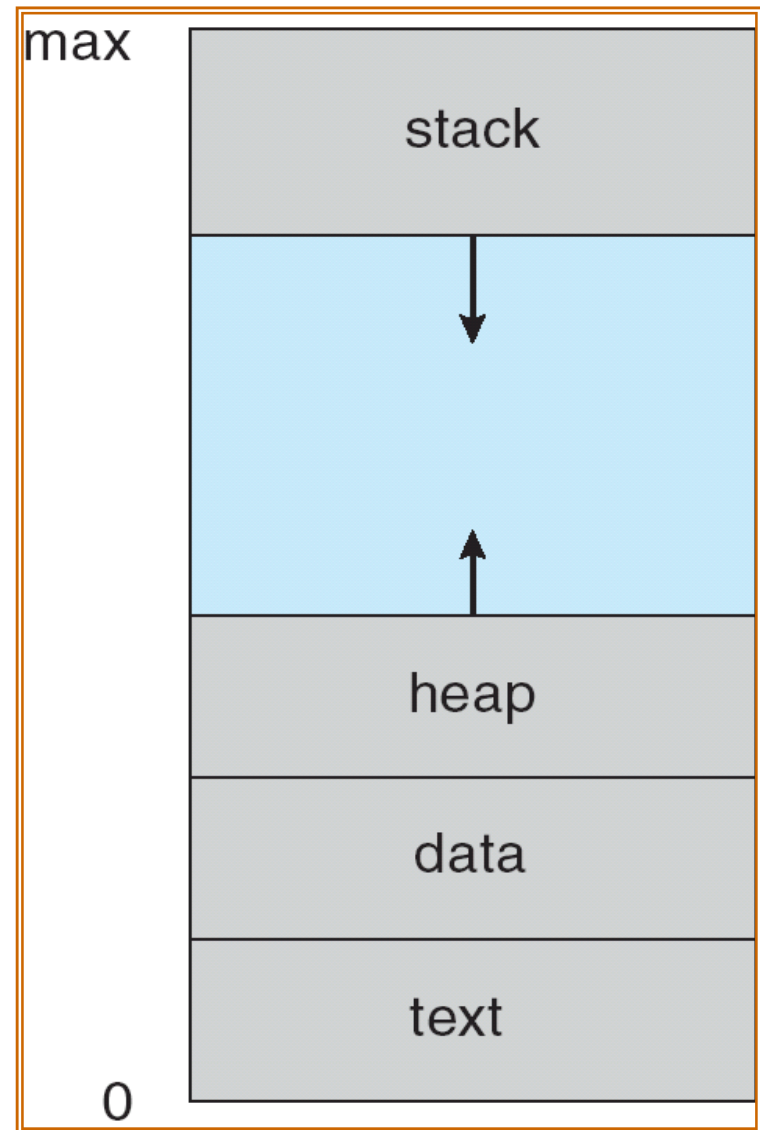


Where the variables below are allocated from?

```
int i;
```

```
int foo(int x)
{
    int *y;

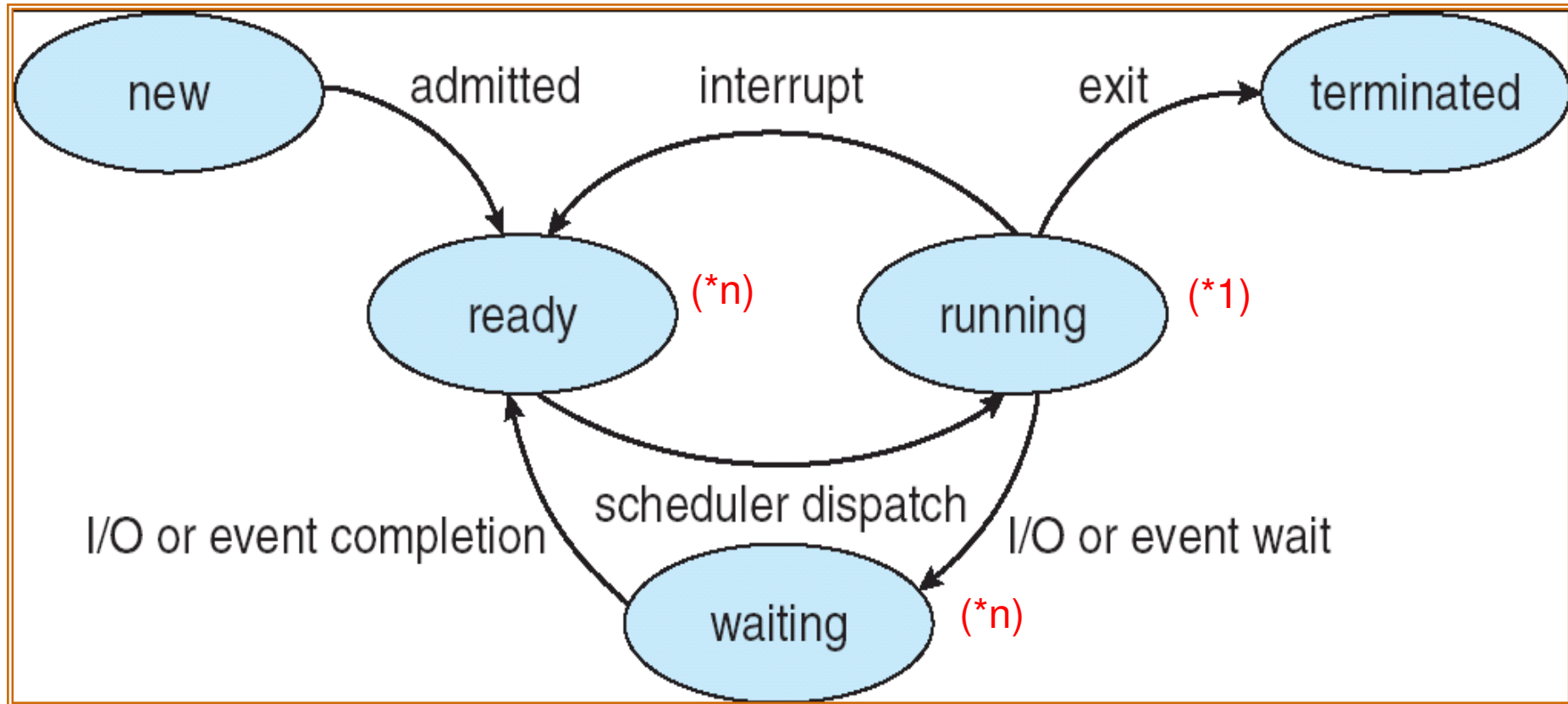
    i=0;
    y=(int *)malloc(100);
}
```



# Process State

- As a process executes, it changes state
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **ready**: The process is waiting to be assigned to a processor
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution

# Diagram of Process State



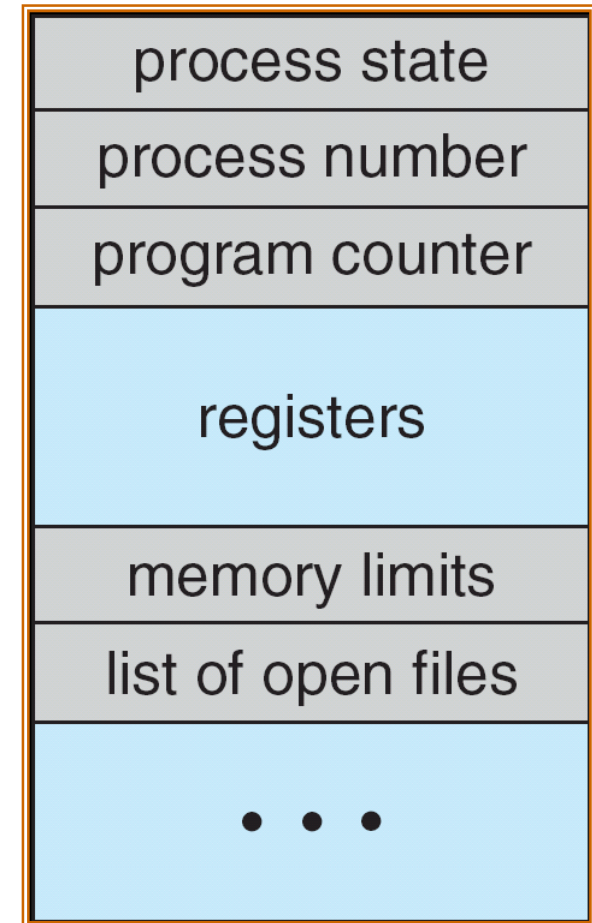


- A running process **voluntarily** leaves the running state
- Running → waiting: the running process requests a system service that can not be immediately fulfilled
  - Triggered by traps (calling the kernel for I/O service)
- When a running process **involuntarily** leaves the running state
- Running → ready: the running process runs out its time quota under time sharing.
  - Triggered by timer interrupts
- Running → ready, case 2: I/O interrupts make a high-priority process ready and the running process is preempted by the high-priority process
  - Triggered by I/O interrupts

- Hardware interrupts can trigger which one(s) of the following transitions?
  1. Running → ready
  2. Running → waiting
  3. Waiting → ready
- What is the state transition of
  - Starting an synchronous I/O
  - Process resume
  - Process suspend?

# Process Control Block (PCB)

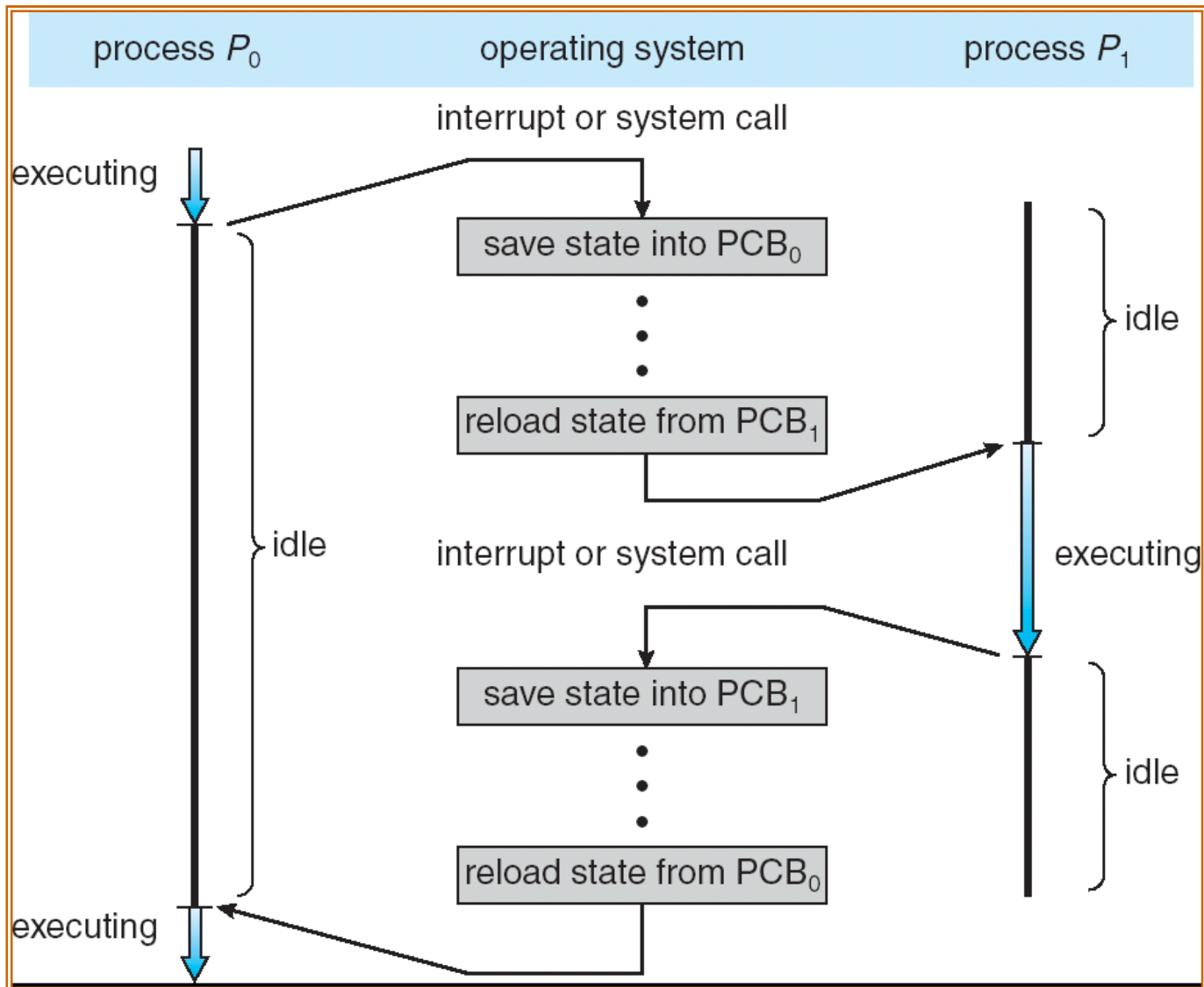
- Information associated with each process
  - Process **state**
  - Saved CPU registers values
  - CPU scheduling info (e.g., **priority**)
  - Memory-management information (e.g., **segment table and page-table base register**)
  - I/O status info (e.g., **opened files**)
  - Etc



# Context Switch

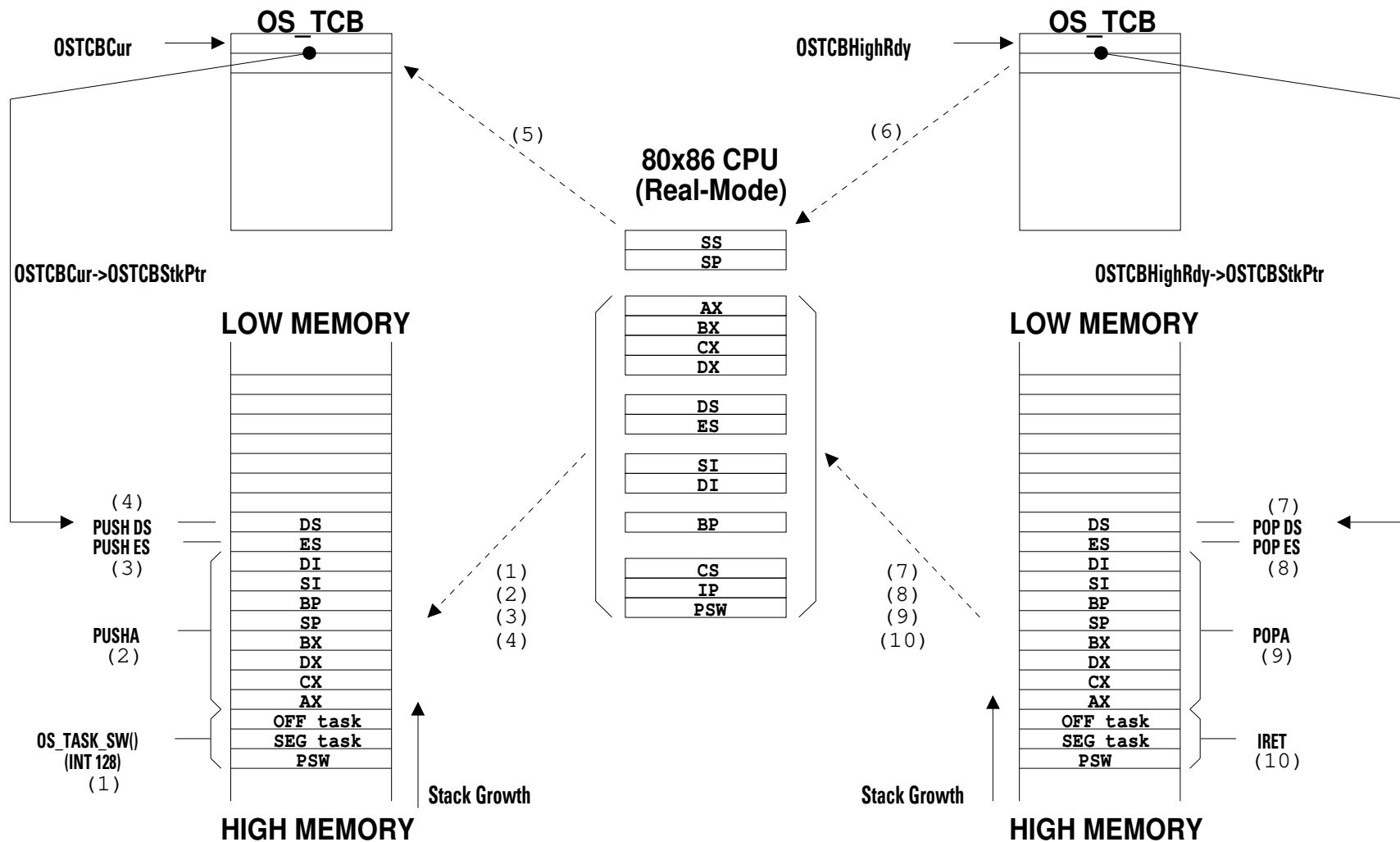
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is an overhead; the system does no useful work while switching
- Time dependent on hardware
  - Roughly 2000 ns/cxtsw on Intel 5150 (2.66 GHz)
  - And the subsequent costs of pipeline stall and cache pollution

# CPU Switch From Process to Process



# Example: Context Switch in uC/OS-2

```
*****
;
*
;
;           PERFORM A CONTEXT SWITCH (From task level)
;           void OSCtxSw(void)
;
; Note(s): 1) Upon entry,
;           OSTCBCur   points to the OS_TCB of the task to suspend
;           OSTCBHighRdy points to the OS_TCB of the task to resume
;
;           2) The stack frame of the task to suspend looks as follows:
;
;           SP -> OFFSET of task to suspend   (Low memory)
;                SEGMENT of task to suspend
;                PSW      of task to suspend   (High memory)
;
;           3) The stack frame of the task to resume looks as follows:
;
;           OSTCBHighRdy->OSTCBStkPtr --> DS                (Low memory)
;                                           ES
;                                           DI
;                                           SI
;                                           BP
;                                           SP
;                                           BX
;                                           DX
;                                           CX
;                                           AX
;                                           OFFSET of task code address
;                                           SEGMENT of task code address
;                                           Flags to load in PSW                (High memory)
*****
;
```



```

_OSctxSw  PROC  FAR
;
        PUSHA                ; Save current task's context
        PUSH  ES              ;
        PUSH  DS              ;
;
        MOV  AX, SEG _OSTCBCur ; Reload DS in case it was altered
        MOV  DS, AX           ;
;
        LES  BX, DWORD PTR DS:_OSTCBCur ; OSTCBCur->OSTCBStkPtr = SS:SP
        MOV  ES:[BX+2], SS      ;
        MOV  ES:[BX+0], SP      ;
;
        CALL FAR PTR _OSTaskSwHook ; Call user defined task switch hook
;
        MOV  AX, WORD PTR DS:_OSTCBHighRdy+2 ; OSTCBCur = OSTCBHighRdy
        MOV  DX, WORD PTR DS:_OSTCBHighRdy  ;
        MOV  WORD PTR DS:_OSTCBCur+2, AX     ;
        MOV  WORD PTR DS:_OSTCBCur, DX       ;
;
        MOV  AL, BYTE PTR DS:_OSPrioHighRdy ; OSPrioCur = OSPrioHighRdy
        MOV  BYTE PTR DS:_OSPrioCur, AL     ;
;
        LES  BX, DWORD PTR DS:_OSTCBHighRdy ; SS:SP = OSTCBHighRdy->OSTCBStkPtr
        MOV  SS, ES:[BX+2]                  ;
        MOV  SP, ES:[BX]                    ;
;
        POP  DS                             ; Load new task's context
        POP  ES                             ;
        POPA                             ;
;
        IRET                                ; Return to new task
;
_OSctxSw  ENDP

```

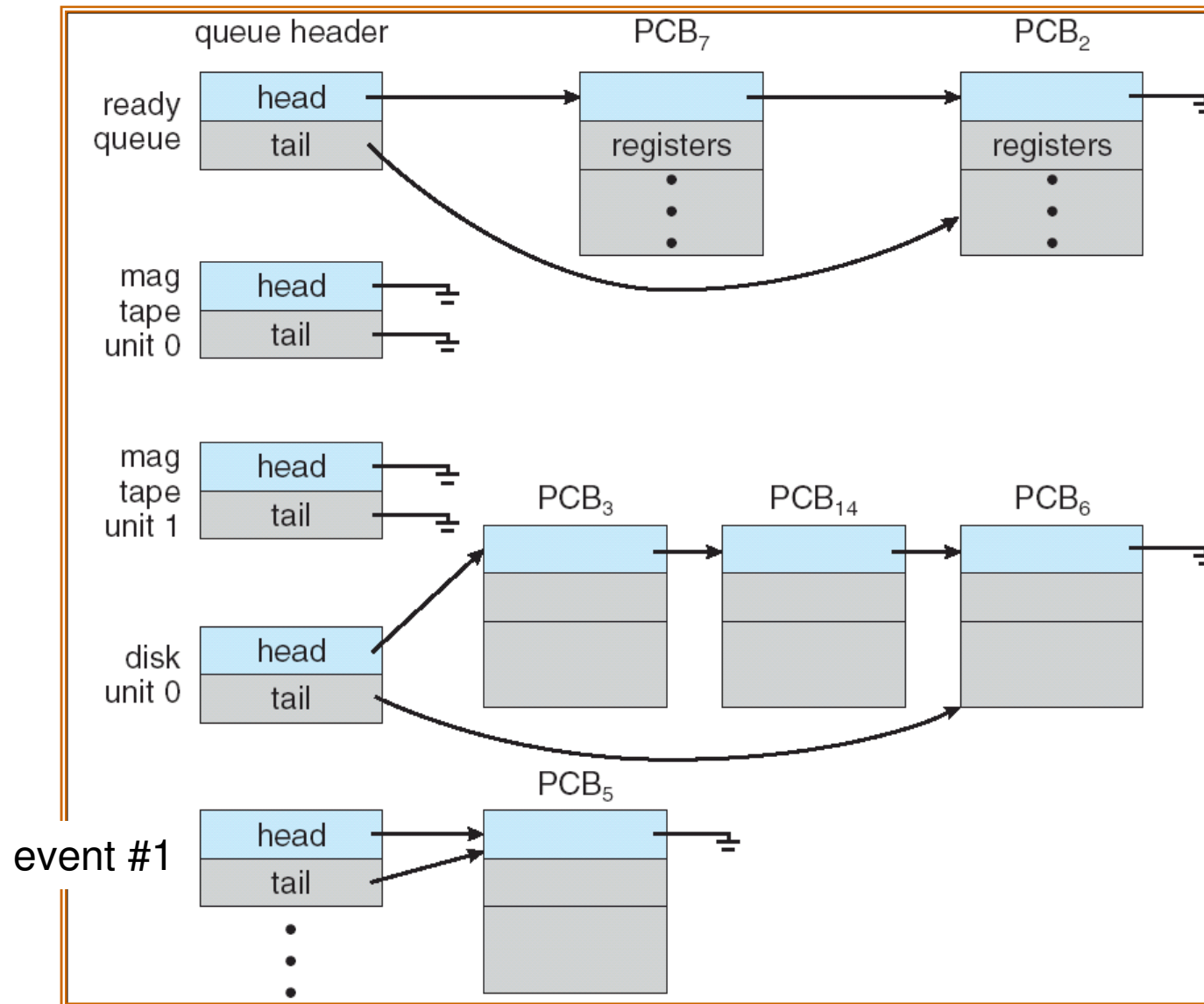


# PROCESS SCHEDULING

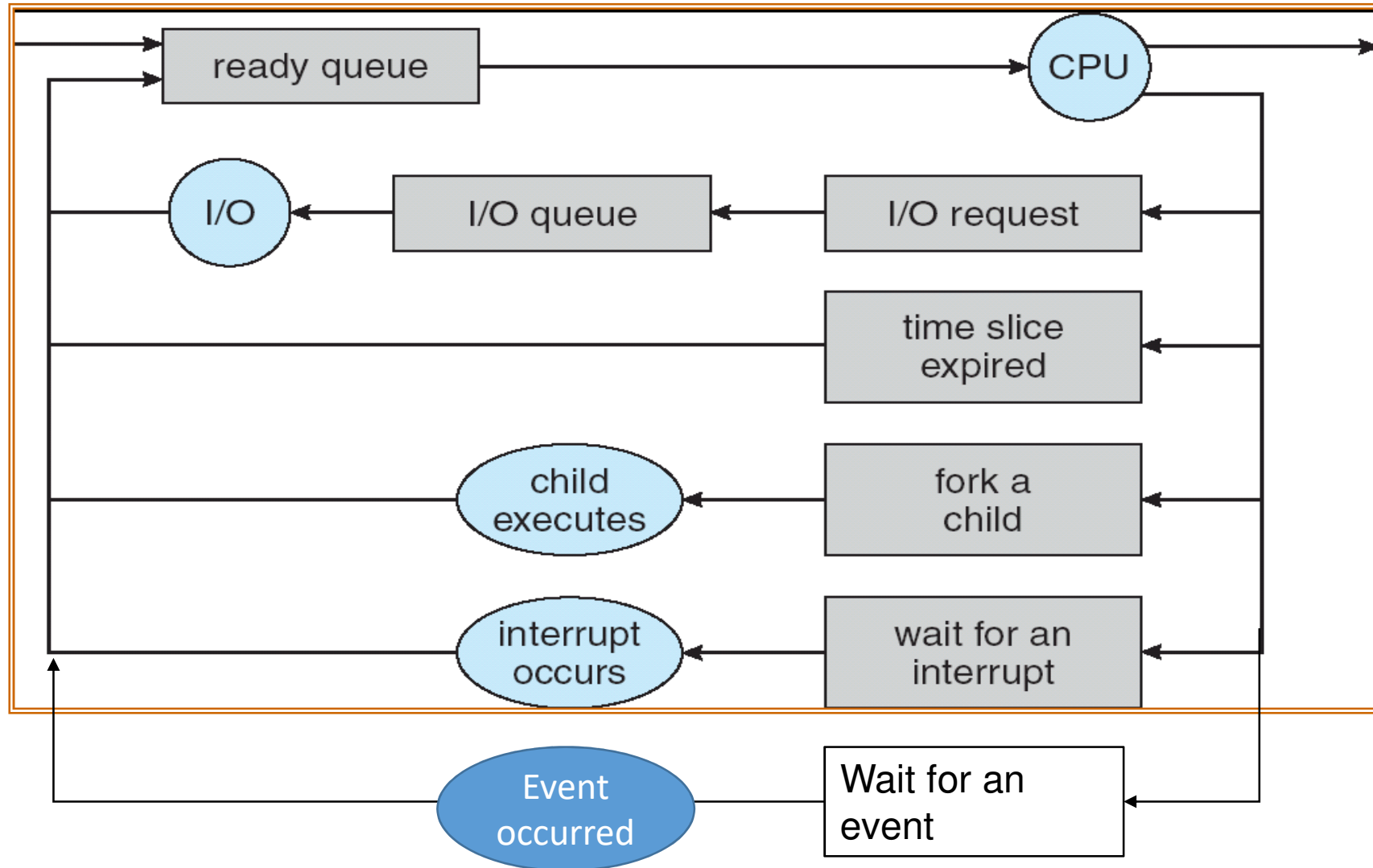
# Process Scheduling Queues

- **Ready** queue – set of all processes residing in main memory, ready for execution
- Device queues – set of processes **waiting** for an I/O device
- Event queues – set of processes **waiting** for an event (e.g., semaphore)
- Processes migrate among the various queues

# Various Process Queues



# Representation of Process Scheduling



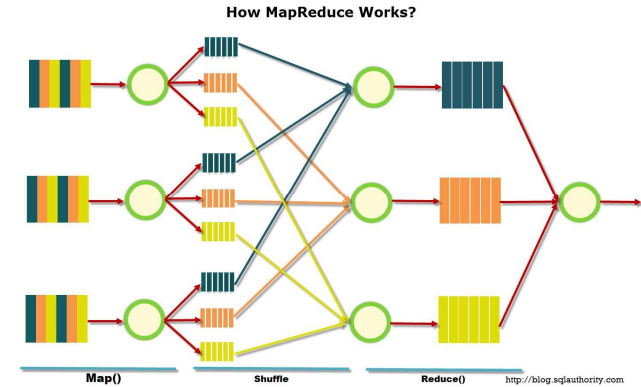
# Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

## Schedulers (Cont.)

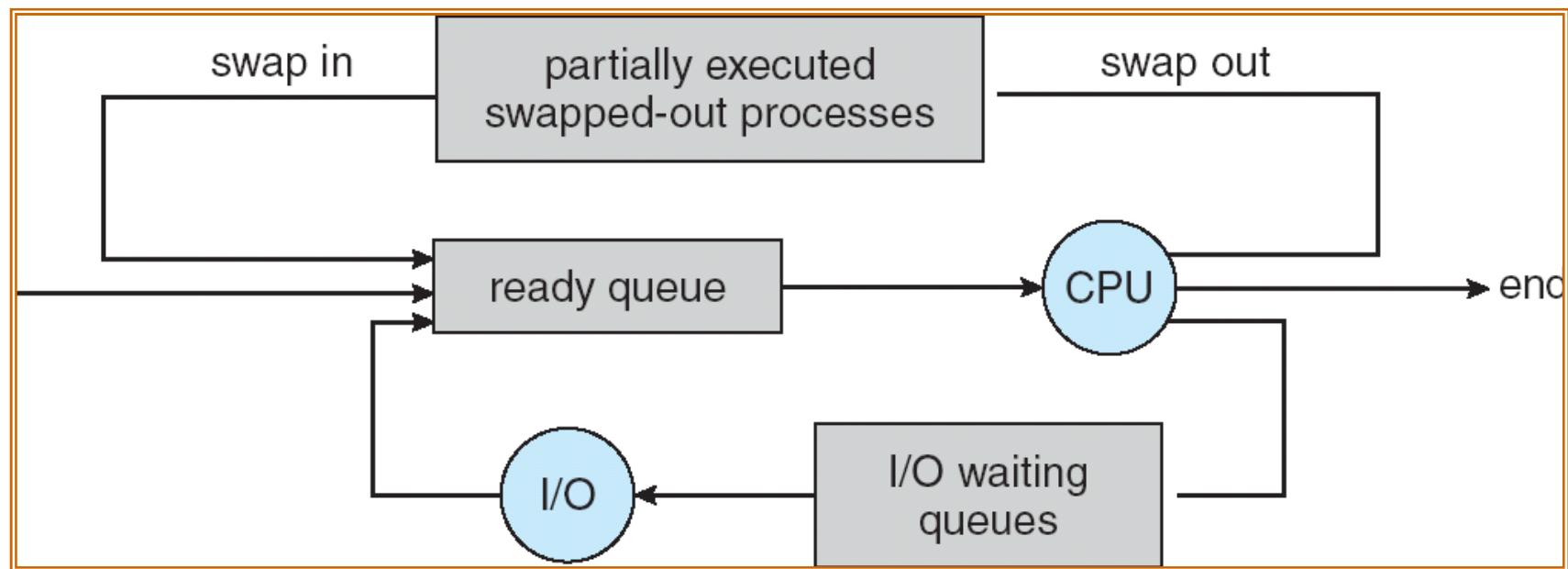
- Short-term scheduler is invoked very frequently (milliseconds)  $\Rightarrow$  (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes)  $\Rightarrow$  (may be slow)
- The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

# Long-Term Scheduler



- In batch-processing systems, the long-term scheduler is to make a good mix of I/O bound processes and CPU-bound processes
  - Modern batch processing example: MapReduce
- Timesharing systems do not have long-term schedulers
  - The degree of multiprogramming is limited by physical limitation (e.g., RAM space)
  - The user will give up if the system cannot launch any more processes

# Addition of Medium Term Scheduling



Swapping out: “saving” the memory image of a process (to a disk) to give memory space to new processes

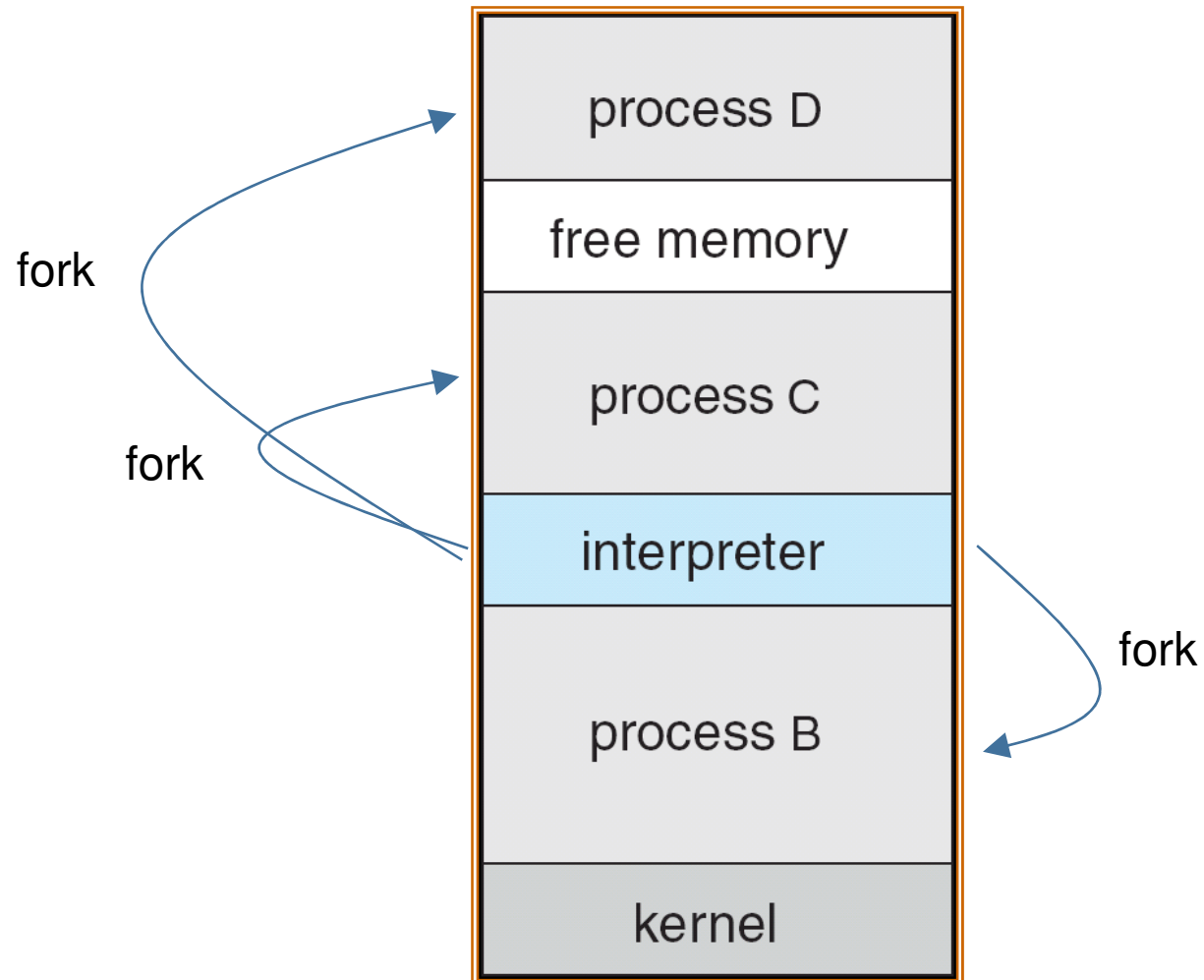


# Checklist

- Long term scheduler
- Short term scheduler
- Mid term scheduler
- I/O-bound and CPU-bound processes

# OPERATIONS ON PROCESSES (CREATION & TERMINATION)

# Physical Memory Layout of a Multiprogramming System (No Paging)



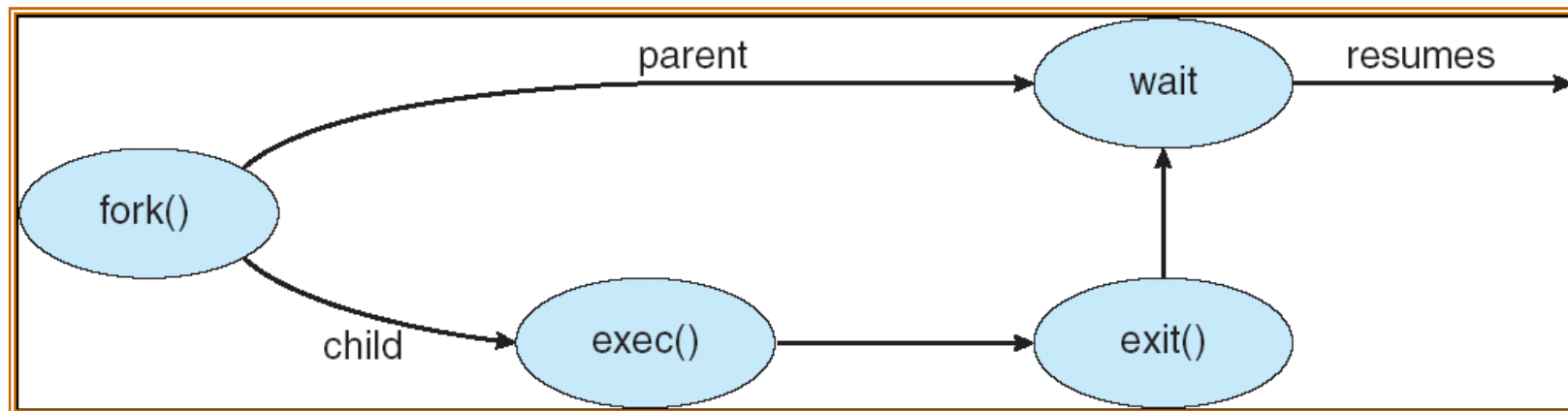
# Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

## Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process (clone the calling process)
  - **exec** system call used after a fork to replace the process' memory space with a new program
- Right after fork():
  - The child is an **exact copy** of the parent

# Process Creation



# C Program Forking Separate Process (in UNIX)

```
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```

Parent and child see different return values!!!

The child won't return here after exec()

The parent has child's pid so it can kill the child (if necessary)

# Address Spaces of Parent and Child Processes

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

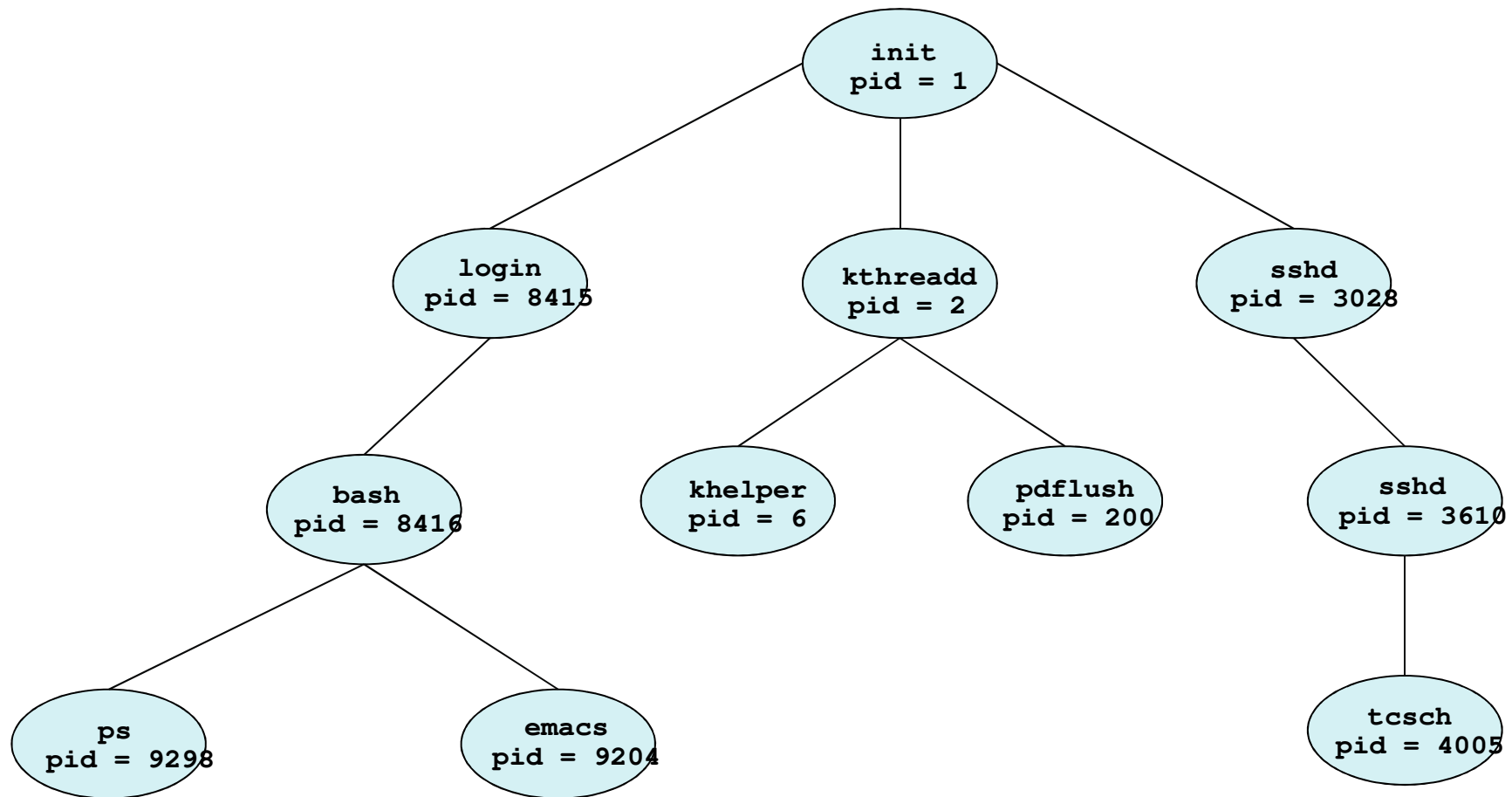
int x=0;

int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        x++;
        exit(0);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("%d",x);
        exit(0);
    }
}
```

What is the output of this program?



# A Tree of Processes in Linux



# Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait() )
  - Synchronous termination
- Parent may terminate execution of its children
  - Sending a signal (SIGKILL) to a child
  - Asynchronous termination

# Orphan Processes and Zombie Processes

- A zombie (defunct) process
  - A process that has terminated (all resources released) but its return value has not been retrieved by its parent yet
  - It will not be deleted from the process table (!)
- An orphan process
  - A process whose parent process has terminated
  - Linux: an orphan will be adopted by process 0 (init)
  - Process 0 (init) will wait (/retrieve the return value) of an orphan
- Zombie implies orphan? Orphan implies zombie?

# A Zombie Child Process

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

main(){

    if(fork()==0){
        // child process
        printf("child pid=%d\n", getpid());
        exit(0)
    }

    // parent process
    sleep(20); // let the child print the message
    printf("parent pid=%d \n", getpid());
    exit(0);
}
```

After the child terminates, it becomes a zombie until being adopted and read by init.

## fork() then exec()

- fork() requires to make a copy of the current process, but the following exec() replaces the address space
- The copying is efficiently implemented through memory mapping, with the assistance of MMU
- Use vfork() instead of fork() if the CPU is not equipped with an MMU

## vfork(): parent and child share most resources

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

int x=0;

int main()
{
    pid_t pid;
    /* fork another process */
    pid = vfork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        x++; // often, here calls exec()
        _exit(0);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("%d",x);
        exit(0);
    }
}
```

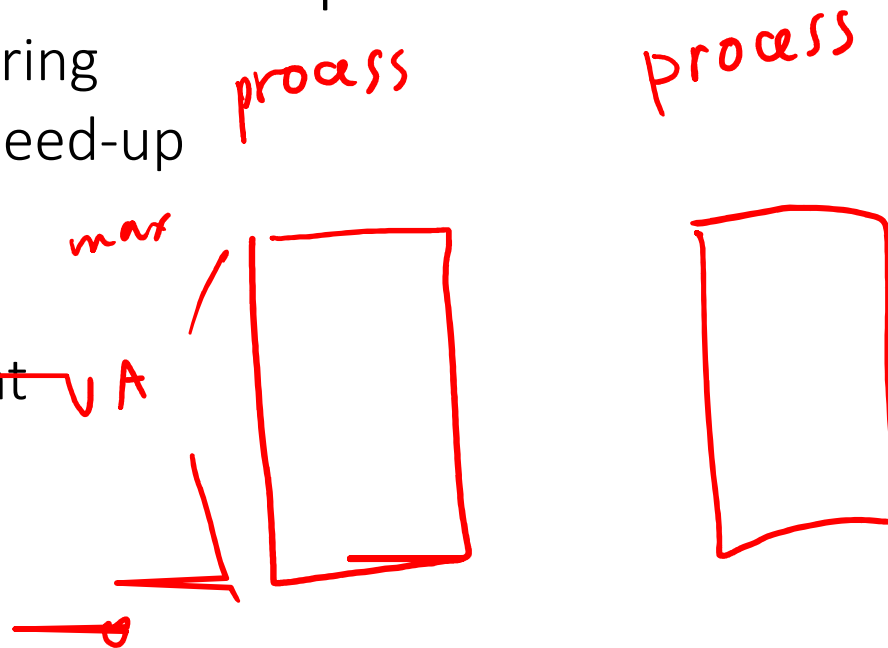
What is the output of this program?

# INTER-PROCESS COMMUNICATION

# Cooperating Processes

- Advantages of process cooperation

- Information sharing
- Computation speed-up
- Modularity
- Convenience
- UX improvement



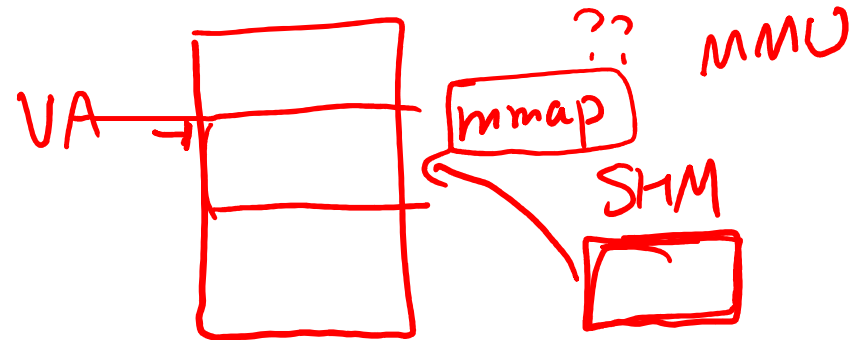


known in physics



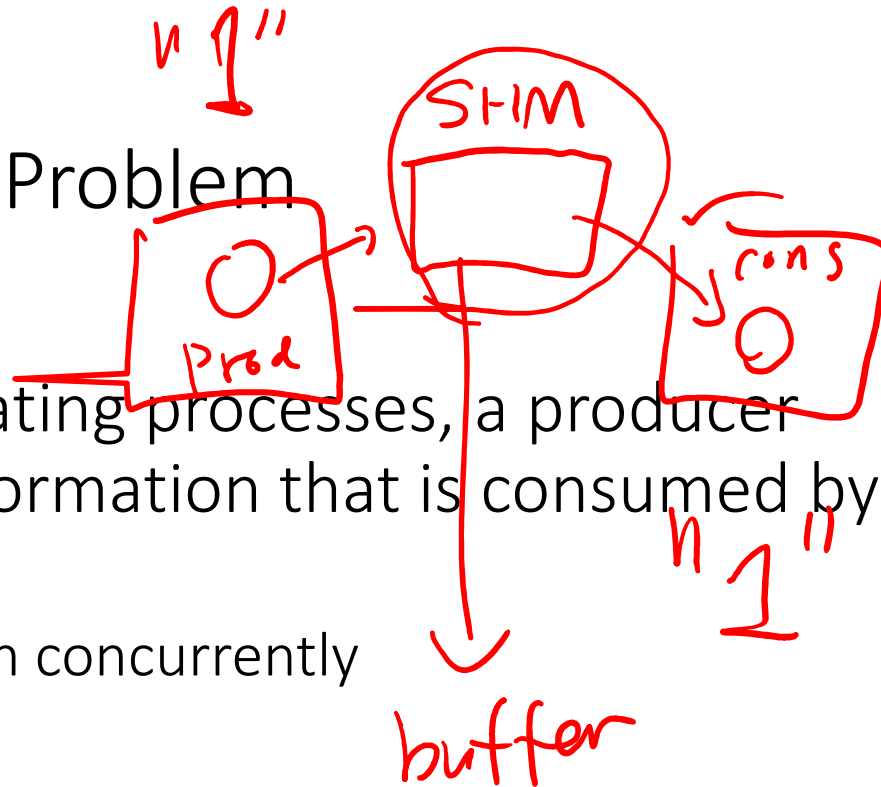
# IPC- SHARED MEMORY

# Shared Memory



- Linux offers the following system calls for shared memory management
  - `shmget()` – create a block of shared memory
  - `shmat()` – attach shared memory to the current process's address space
  - `shmdt()` – detach shared memory from the current process's address space
  - `shmctl()` – control shared memory (including delete)
- Let assume that a piece of shared memory has been setup between two processes

# Producer-Consumer Problem



- Paradigm for cooperating processes, a producer process produces information that is consumed by a consumer process
  - The two processes run concurrently
- Objective:
  - to synchronize a producer and a consumer via **shared memory**
- Issues:
  - **The buffer size is limited**
  - **Overwriting and null reading are not allowed**

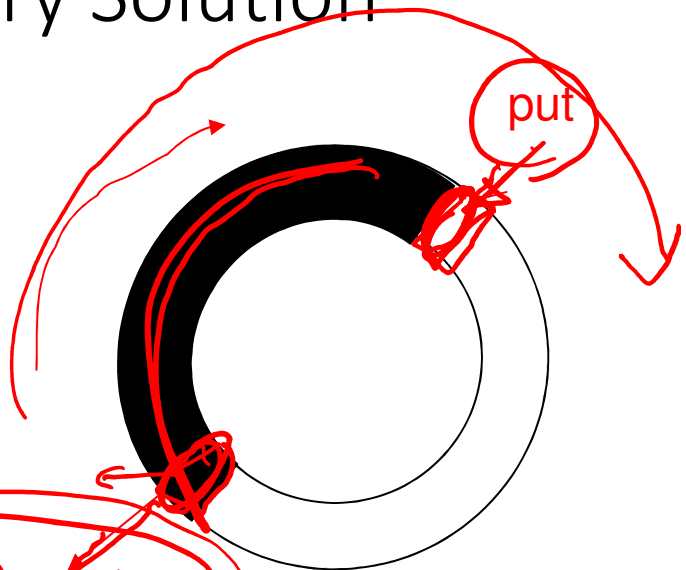
# Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
```

```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

SMM



Circular (ring) buffer



empty  
 $P == C$

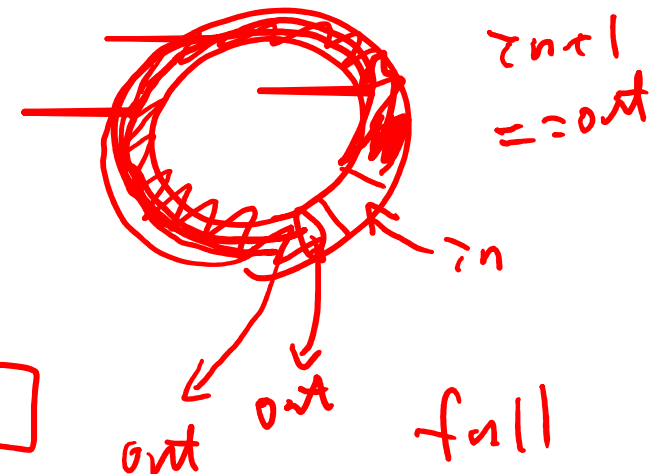
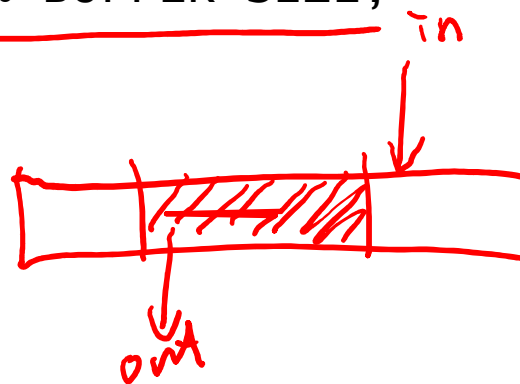
$(P+1) \% N$   
full == C

- Solution is correct, but can only use  $BUFFER\_SIZE - 1$  elements

- What are the conditions for **buffer full** and **buffer empty**?

## Bounded-Buffer – Insert() Method

```
while (true) {  
    prod Free == 0 (full)  
    /* Produce an item */  
    while ((in + 1) % BUFFER SIZE count) == out) full  
        ; /* do nothing -- no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```



## Bounded Buffer – Remove() Method

```
while (true) {
```

```
    while (in == out)
```

```
        // do nothing -- nothing to consume
```

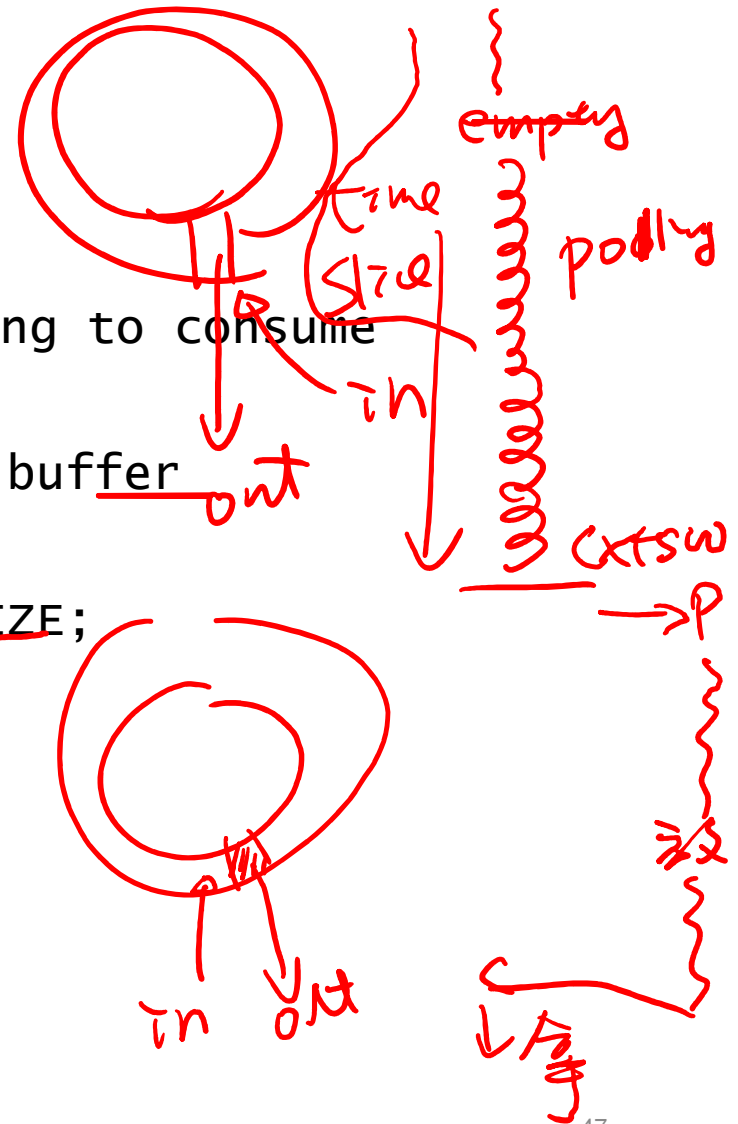
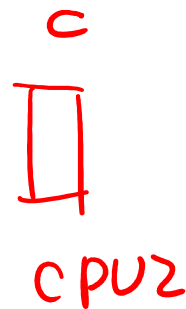
```
        // remove an item from the buffer
```

```
        item = buffer[out];
```

```
        out = (out + 1) % BUFFER_SIZE;
```

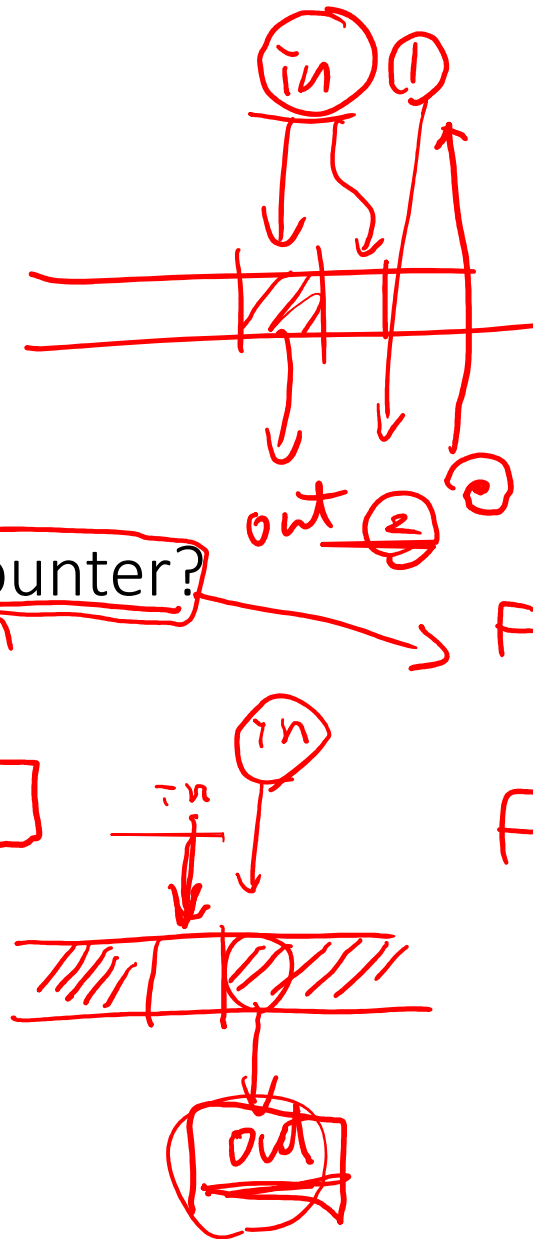
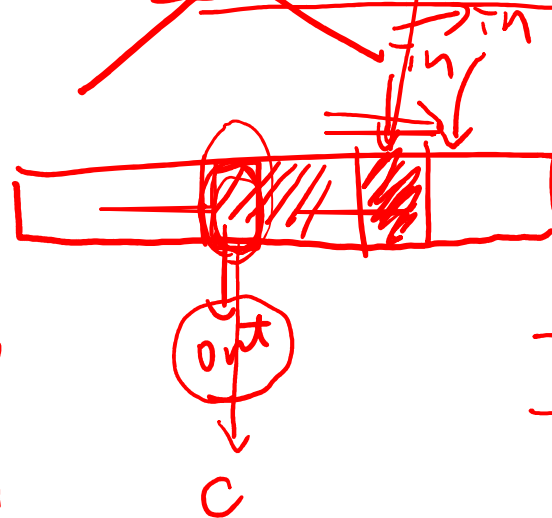
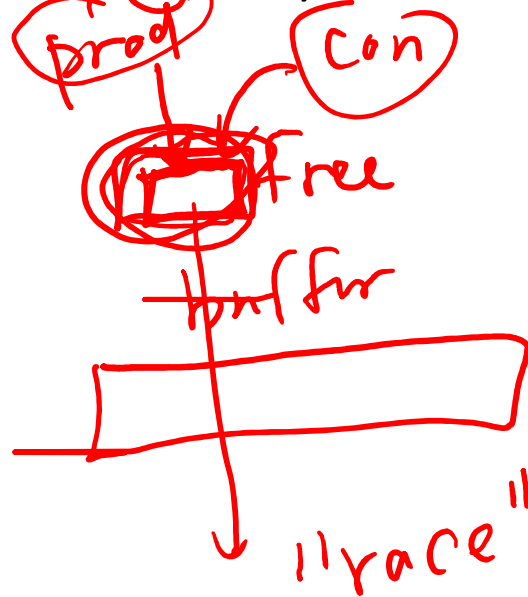
```
        return item;
```

```
}
```



# Bounded Buffer Problem

- ~~Data corruption?~~
- ~~Performance issue?~~
- ~~Why not to use a free-slot counter?~~



~~Free = 0~~  
full

Free = N  
empty



# IPC- MESSAGE PASSING

# Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other **without resorting to shared variables**
- IPC facility provides two operations:
  - ~~send~~(message) – message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

# Interprocess Communication (IPC)

- Messages can be buffered in the link

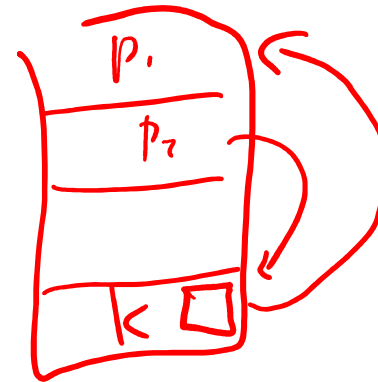
•  $P \rightarrow [\text{link <buffer>}] \rightarrow Q$

*blocked* *full*

*blocked* *Run*  
*Wait*

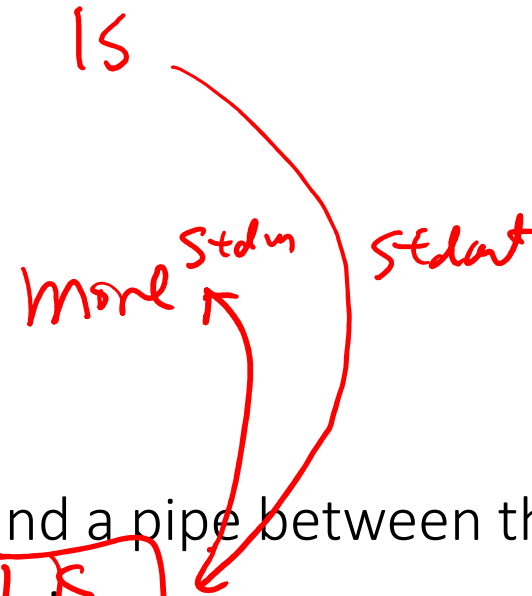
- P will be *blocked* on sending if the link buffer is full
- Q will be *blocked* on receiving if the link buffer is empty

*synchronize*



## Example: Linux Pipe

- A basic mechanism for IPC
  - Widely used, e.g., `"ls | more"`
  - A process "ls", a process "more", and a pipe between them
- The `system call pipe()` creates a pipe
  - Receiver must close the output side, and receives from the input side
  - Sender must close the input side, and write to the output side
  - A pipe is created and configured by the parent process



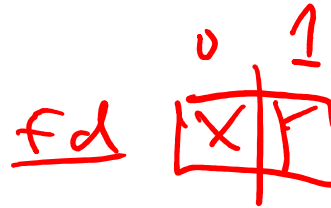
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
int main(void)
{
```

```
    int fd[2]; nbytes;
    pid_t childpid;
    char string[] = "Hello, world!\n";
    char readbuffer[80];
```

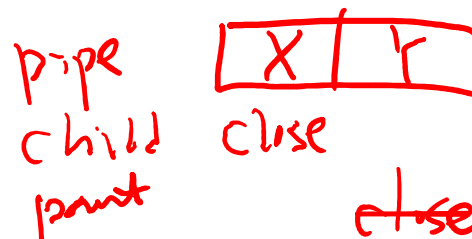
```
    pipe(fd); // create the pipe before calling fork()
```

```
    if((childpid = fork()) == -1)
    {
        perror("fork");
        exit(1);
    }
```



fork(1);

fd (in) (out)



if (childpid == 0)

/\* Child process closes up input side of pipe \*/  
close(fd[0]);

child

/\* Send "string" through the output side of pipe \*/  
write(fd[1], string, (strlen(string)+1));  
exit(0);

"Hello World"

}  
else  
{

/\* Parent process closes up output side of pipe \*/  
close(fd[1]);

/\* Read in a string from the pipe \*/  
nbytes = read(fd[0], readbuffer, sizeof(readbuffer));  
printf("Received string: %s", readbuffer);

}

return(0);

}

"He ..."

# UNIX Signals

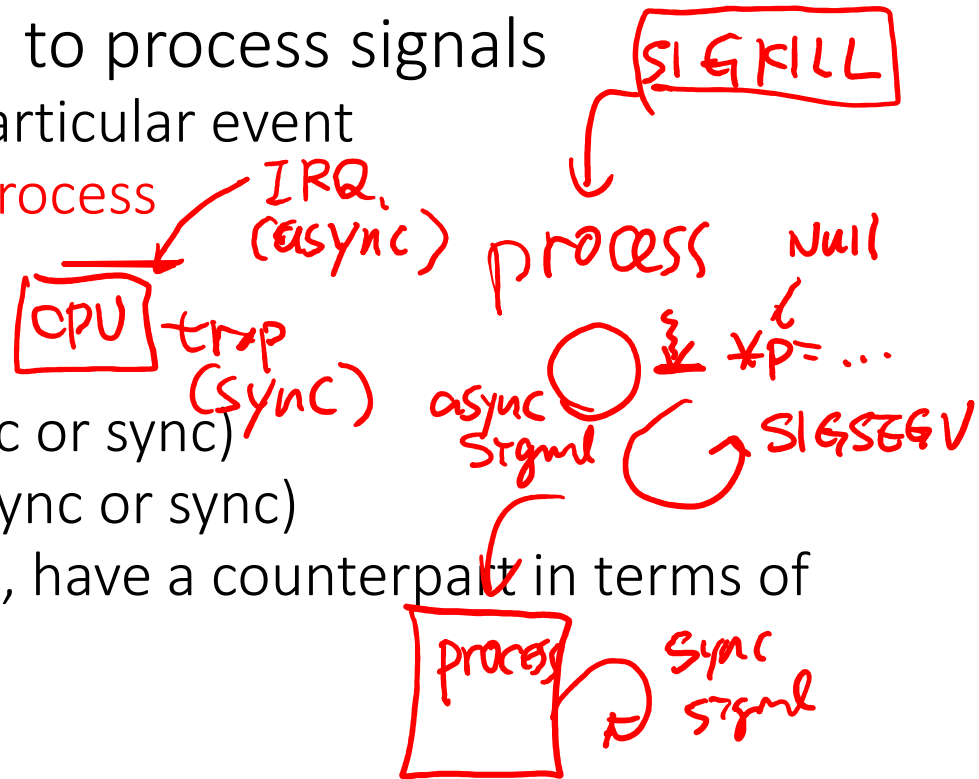
1 1 : 2 0

default handlers

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a **process**
  - Signal is handled

## Analogy

- Interrupts for CPU (async or sync)
- Signals for processes (async or sync)
- Many signals, but not all, have a counterpart in terms of interrupts



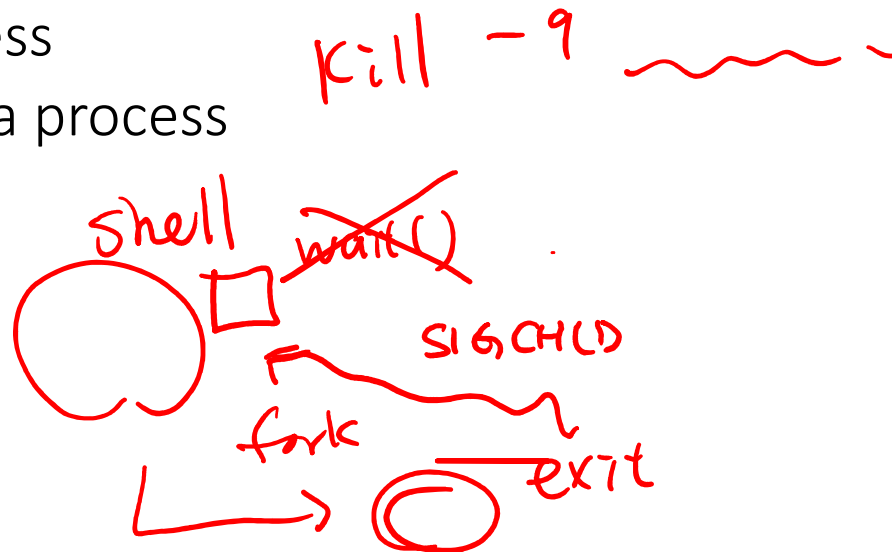
# Signal Handling

- Synchronous signals
  - A signal that is delivered to the process caused the event
  - E.g., divide overflow and memory-access violations
- Asynchronous signals
  - A signal that is delivered to a process other than the signaling process
  - E.g., the kill signal
- Signal handlers
  - Default handlers
  - User-defined handlers (using signal() or sigaction())



# UNIX Signal Example

- Synchronous signals
  - SIGSEGV : Memory protection fault
  - SIGFPE : Arithmetic fault, including divided by zero
- Asynchronous signals
  - SIGKILL : Kill a process
  - SIGSTOP : Suspend a process
  - SIGCHLD: ???



# Handling SIGSEGV on your own

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

void sigsegv_handler(int sig) {
    printf("Received segmentation violation (SIGSEGV). \n");
    exit(0);
}

int main() {
    int *null_pointer=(int *)NULL;
    signal(SIGSEGV, sigsegv_handler);

    printf("About to segfault:\n");
    *null_pointer=0;
    printf("Shouldn't be here!\n");
    return 1;
}
```

default → exit(0);

handle

SIGSEGV ←

MMU

Trap ↓ OS

~~X~~

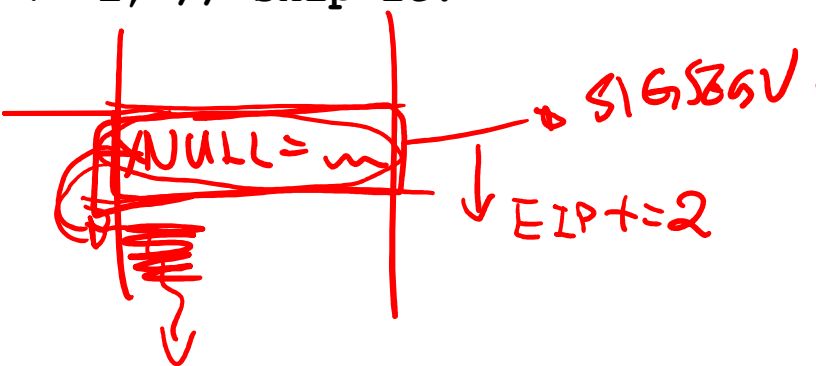
# Handling SIGSEGV on your own

```
void action(int sig, siginfo_t* siginfo, void* context)
{
    sig=sig; siginfo=siginfo;

    // get execution context
    mcontext_t* mcontext = &((ucontext_t*)context)->uc_mcontext;

    uint8_t* code = (uint8_t*)mcontext->gregs[REG_EIP];
    if (code[0] == 0x88 && code[1] == 0x10) { // mov %dl, (%eax)
        mcontext->gregs[REG_EIP] += 2; // skip it!
        return;
    }
}

main()
{
    ...
    sigaction(SIGSEGV, ...);
    ...
    for (int i = 0; i < 10; i++) { ((unsigned char*)0)[i] = i; }
}
```



End of Chapter 3