東南大學
SOUTHEAST UNIVERSITY

OPERATING SYSTEM CONCEPTS

Chapter 3. Processes

A/Prof. Kai Dong

Warm-up

What Is the Output?

```c
/* cpu.c */
int main(int argc, char *argv[]) {
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];
        while (1) {
                spin(1);
                printf("%s\n", str);
        }
        return 0;
}
```

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu A
```

Warm-up

What Is the Output?

```c
/* cpu.c */
int main(int argc, char *argv[]) {
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];
        while (1) {
                spin(1);
                printf("%s\n", str);
        }
        return 0;
}
```

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu A
```

```
A
A
A
A
^C
prompt>
```

Warm-up

What Is the Output? (contd.)

```c
/* cpu.c */
int main(int argc, char *argv[]) {
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];
        while (1) {
                spin(1);
                printf("%s\n",str);
        }
        return 0;
}
```

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
```

Warm-up

What Is the Output? (contd.)

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
```

```
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Warm-up

- How to implement virtualization of the CPU? — The OS will need both some low-level machinery (mechanisms) as well as some high-level intelligence (policies).
- The time-sharing mechanism (context switch) + scheduling policy — By running one process, then stopping it and running another, and so forth.
- The **process** (or job) is one of the most fundamental abstractions that the OS provides to users.

Objectives

- To introduce the notion of a process – a program in execution, which forms the basis of all computation.
- To describe the various features of processes, including scheduling, creation and termination, and communication.
- To explore interprocess communication using shared memory and message passing.
- To describe communication in client-server systems.

Contents

Contents

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* and *process* almost interchangeably
- **Process** — a program in execution; process execution must progress in sequential fashion
- Multiple parts (machine state of the running program)
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
    - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

Process Concept

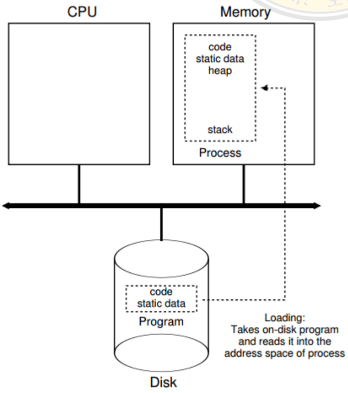
Process Vs. Program

- Program is *passive* entity stored on disk (executable file), process is *active*
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process Concept

Loading: From Program To Process

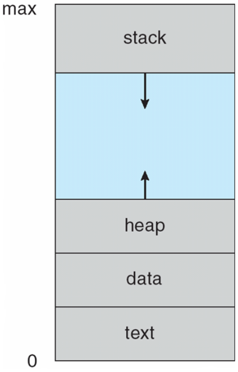
- How programs are transformed into processes?
  - *Loading* (*eagerly* or *lazily*)
  - *Paging* and *swapping*
  - Allocation of stack and heap
  - IO initialization



Process Concept

Process in Memory

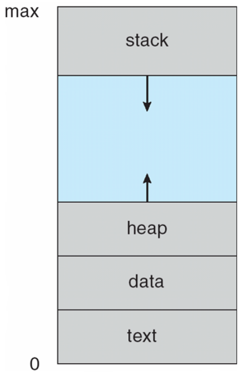
```c
/* main.c */
int a = 0;
char *p1;
int main(int argc, char *argv[]) {
        int b;
        char s[] = "abc";
        char *p2;
        char *p3 = "123456";
        p1 = (char *)malloc(10);
        p2 = (char *)malloc(20);
        return 0;
}
```



Process Concept

Process in Memory

```c
/* main.c */
int a = 0; // data
char *p1;
int main(int argc, char *argv[]) {
        int b;
        char s[] = "abc"; // stack
        char *p2;
        char *p3 = "123456"; // stack
        p1 = (char *)malloc(10); // heap
        p2 = (char *)malloc(20); // heap
        return 0;
}
```



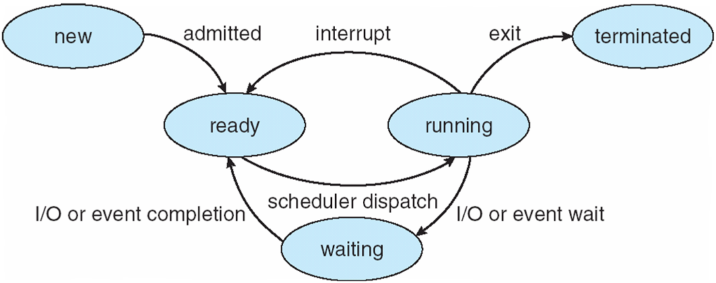
Process Concept

Process State

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

Process Concept

Diagram of Process State



Process Concept

Process Data Structure

- Review: Motivating processes, why not programs?
  - Can a program be stopped and then be run again?
    - No, unless we can record the machine state of the running program.
    - A process can be viewed as a running program with machine states.
- OS is a program, so it has some key data structures that *track the state of each process*.
  - Process lists for all ready / running / waiting processes.
  - What else? — Types of information an OS needs to track processes.
- **Process Control Block** — An example: xv6 kernel

Process Concept

Process Control Block

```c
// the registers
struct context {
        int eip; // Program counter / instruction pointer
        int esp, ebx, ecx, edx, esi, edi, ebp; // Other registers
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
struct proc {
        char *mem; // Start of process memory
        uint sz; // Size of process memory
        char *kstack; // Bottom of kernel stack for this process
        enum proc_state state; // Process state
        int pid; // Process ID
        struct proc *parent; // Parent process
        void *chan; // If non-zero, sleeping on chan
        int killed; // If non-zero, have been killed
        struct file *ofile[NOFILE]; // Open files
        struct inode *cwd; // Current directory
        struct context context; // Switch here to run process
        struct trapframe *tf; // Trap frame for the current interrupt
};
```

Process Concept

Process Execution — Direct Execution

- Suppose the following direct execution protocol.
- Any problem?

OS (kernel mode)	**Program**
create entry for process list	
allocate memory for program	
load program into memory	
set up stack with argc/argv	
clear registers	
execute call *main*()	
	run *main*()
	execute return from main
free memory of process	
remove from process list	

Process Concept

Process Execution — Limited Direct Execution

- Direct execution is fast, but
- Two problems with direct execution
- Protection
  - How can the OS make sure the program doesn't do anything that we don't want it to do?
  - Protection via dual mode and system call.
- Time sharing
  - How does the operating system stop it from running and switch to another process?
  - Time sharing via context switch.

Process Concept

Process Execution — Limited Direct Execution

OS @boot (kernel mode)	**Hardware**
initialize *trap table*	
	remember addresses of …
	system call handler
	timer handler
	illegal instruction handler
start interrupt timer	
	start timer; interrupt after X ms

Process Concept

Process Execution — Protection

OS @run (kernel mode)	**Hardware**	**Program (user mode)**
create entry for process list		
allocate memory for program		
load program into memory		
set up user stack with argv		
fill kernel stack with reg/PC		
return-from-trap		
	move to *user mode*	
		run main()
		...
		call *system call*
		trap into OS

to be contd.

Process Concept

Process Execution — Protection (contd.)

OS @run (kernel mode)	**Hardware**	**Program (user mode)**
		call *system call*
		trap into OS
	save regs to kernel stack	
	move to *kernel mode*	
	jump to *system call handler*	
handle *trap*		
return-from-trap		
	restore regs from kernel stack	
	move to *user mode*	
	jump to PC after trap	
		...
		return from main
		trap (via exit())
free memory of process		
remove from *process list*		

Process Concept

Process Execution — Time-sharing

- Switching Between Processes
  - OS regains control of the CPU via the timer interrupt.
  - Discussion: why hardware interrupt instead of software trap?
- Whether to switch is decided by the scheduler in OS.
- Context switch — saving and restoring context.

Process Concept

Process Execution — Time-sharing (contd.)

OS @boot (kernel mode)	**Hardware**
initialize *trap table*	
	remember addresses of ...
	system call handler
	timer handler
	illegal instruction handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

Process Concept

Process Execution — Time-sharing (contd.)

OS @run (kernel mode)	**Hardware**	**Program (user mode)**
		Process *A*
		...
	timer interrupt	
	save regs(*A*) to k-stack(*A*)	
	move to *kernel mode*	
	jump to *timer handler*	
handle *trap*		
call switch() routine		
save regs(*A*) to PCB(*A*)		
restore regs(*B*) from PCB(*B*)		
switch to k-stack(*B*)		
return-from-trap (into *B*)		
	restore regs(*B*) from k-stack(*B*)	
	move to *user mode*	
	jump to *B*'s PC	
		Process *B*
		...

Process Concept

Register Saves/Restores

- Where is the context stored in a context switch?
  - Somewhere in the memory
- Two types of register saves/restores:
- When the (timer) interrupt occurs
  - User registers are implicitly saved by the hardware, into the kernel stack.
- When the OS decides to switch
  - Kernel registers are explicitly saved by the OS, into the PCB in memory.

Process Concept

Some Details

- What if one interrupt occurs during another interrupt or trap handling?
  - Especially when the kernel stack is not saved into PCB in memory.
  - If you understand this problem, you are now thinking about **concurrency** issues.
- Basically, disable interrupts during interrupt processing.
  - More details? In future lectures on concurrency.

Process Concept

Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
  - Multiple threads of control → **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Will be detailed in Ch4

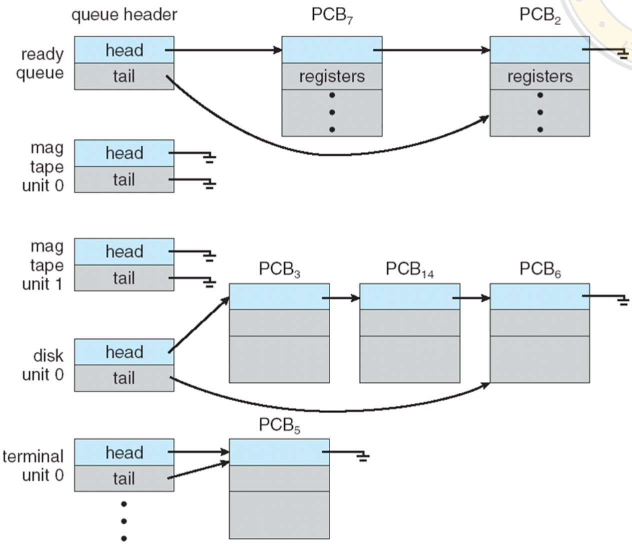
Contents

Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** — set of all processes in the system
  - **Ready queue** — set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** — set of processes waiting for an I/O device
  - Processes migrate among the various queues

Process Scheduling

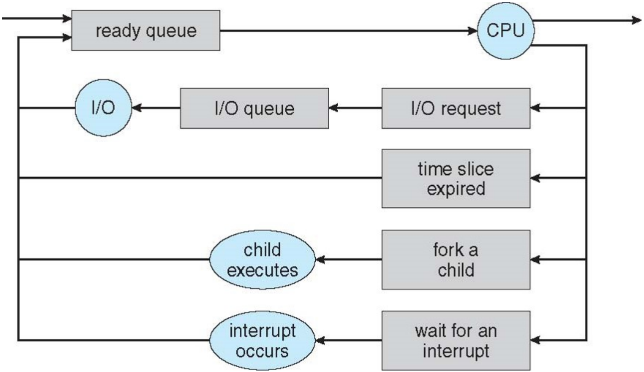
Ready Queue And Various I/O Device Queues



Process Scheduling

Representation of Process Scheduling

- **Queuing** diagram represents queues, resources, flows



Process Scheduling

Schedulers — Short-term Scheduler

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⟶ (must be fast)
- Will be detailed in Chapter 5

Process Scheduling

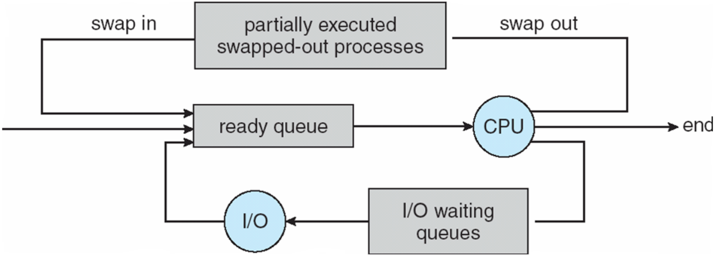
Schedulers — Long-term scheduler

- **Long-term scheduler** (or **job scheduler**) — selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⟶ (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 strives for good process mix

Process Scheduling

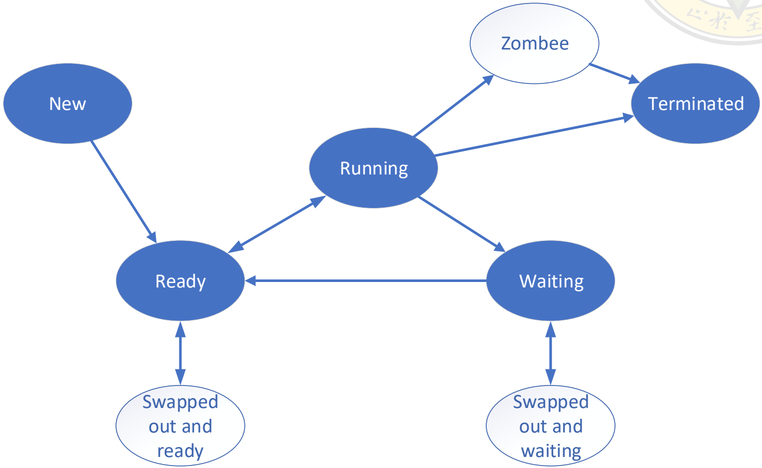
Schedulers — Medium-term scheduler

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Process Scheduling

Diagram of Process State with Schedulers



Contents

Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Operations on Processes

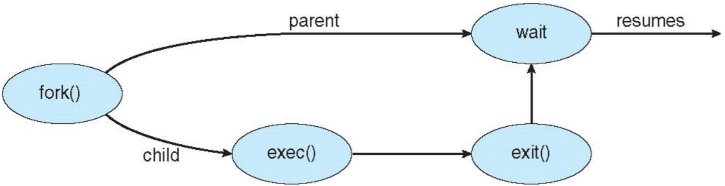
Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Operations on Processes

Process Creation (contd.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - *fork*() system call creates new process
  - *exec*() system call used after a *fork*() to replace the process' memory space with a new program



Operations on Processes

C Program Forking Separate Process

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

int main(int argc, char *argv[]) {
        pid_t pid;
        pid = fork();
        if (pid < 0) { // error occurred
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) { // child process
                execlp("/bin/ls", "ls", NULL);
        }
        else { // parent process
                wait(NULL);
                printf("Child Complete");
        }
        return 0;
}
```

Operations on Processes

Process Termination

- Process executes the last statement and then asks the operating system to delete it using the *exit*() system call.
  - Returns status data from child to parent (via *wait*())
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the *abort*() system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Operations on Processes

Process Termination (contd.)

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - cascading termination. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the *wait*()system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke *wait*()), process is a zombie
- If parent terminated without invoking *wait*(), process is an orphan

Operations on Processes

In Class Exercise

```c
int value = 5;
int main(int argc, char *argv[]) {
        pid_t pid;
        pid = fork();
        if (pid == 0) {
                printf("child process, value1 : %d\n", value);
                value += 15;
                printf("child process, value2 : %d\n", value);
        }
        else if (pid > 0) {
                printf("parent process, value3 : %d\n", value);
                wait(NULL);
                printf("parent process, value4 : %d\n", value);
        }
        exit(0);
}
```

- What is the output?

Operations on Processes

Motivating the APIs

```c
/* p1.c */
int main(int argc, char *argv[]) {
	printf("hello world (pid:%d)\n", (int) getpid());
	int rc = fork();
	if (rc < 0) {
		fprintf(stderr, "fork failed\n");
		exit(1);
	}
	else if (rc == 0) {
		printf ("hello, I am child (pid:%d)\n", (int) getpid());
		char *myargs[3];
		myargs[0] = strdup("wc");
		myargs[1] = strdup("p1.c");
		myargs[2] = NULL;
		execvp(myargs[0], myargs);
		printf("this shouldn't print out");
	}
	else {
		int wc = wait(NULL);
		printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc
			, (int) getpid());
	}
	return 0;
}
```

Operations on Processes

Motivating the APIs (contd.)

```
prompt> ./p1
hello world (pid:29383)
hello, I am child (pid:29384)
        29      107     1030    p1.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

- Motivating The API
- Available to run code after the call to fork() but before the call to *exec*().

```
prompt> wc p3.c > newfile
```

Operations on Processes

Redirection

```c
/* p2.c */
int main(int argc, char *argv[]) {
        printf("hello world (pid:%d)\n", (int) getpid());
        int rc = fork();
        if (rc < 0) {
                fprintf(stderr, "fork failed\n");
                exit(1);
        }
        else if (rc == 0) {
                /********** from here **********/
                close(STDOUT_FILENO);
                open("./p2.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
                /**********  to here   **********/
                char *myargs[3];
                myargs[0] = strdup("wc");
                myargs[1] = strdup("p2.c");
                myargs[2] = NULL;
                execvp(myargs[0], myargs);
                printf("this shouldn't print out");
        }
        else {
                int wc = wait(NULL);
        }
        return 0;
}
```

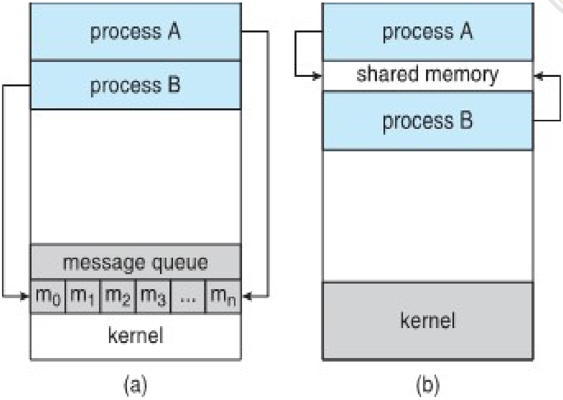
Contents

Interprocess Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Interprocess Communication

Message Passing & Shared Memory



Message passing & Shared memory

Interprocess Communication

Producer-Consumer Problem

- A common paradigm for cooperating processes.
- A producer process produces information that is consumed by a consumer process.
  - A compiler may produce assembly code that is consumed by an assembler.
  - The assembler, in turn, may produce object modules that are consumed by the loader.

Interprocess Communication

Producer-Consumer Problem — Bounded Buffer — Shared-Memory Solution

```c
/* Solution is correct, but can only use BUFFER_SIZE - 1 elements */
#define BUFFER_SIZE 10
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```c
/* producer */
item next_produced;
while (true) {
        while (((in + 1) % BUFFER_SIZE) == out) ;
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

```c
/* consumer */
item next_consumed;
while (true) {
        while (in == out) ;
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

Interprocess Communication

Producer-Consumer Problem — Bounded Buffer — Shared-Memory Solution

in =	0									
out =	0									
i =	0	1	2	3	4	5	6	7	8	9
buffer										

- An exercise for you to provide a solution in which *BUFFER_SIZE* items can be in the buffer at the same time.
- *in* indicates the next production, and
- *out* indicates the next consumption.
- What if $i == out$?
  - *buffer* is empty, or
  - *buffer* is full.

Interprocess Communication

Producer-Consumer Problem — Bounded Buffer — Shared-Memory Solution

in =		1								
out =	0									
i =	0	1	2	3	4	5	6	7	8	9
buffer	*									

- An exercise for you to provide a solution in which *BUFFER_SIZE* items can be in the buffer at the same time.
- *in* indicates the next production, and
- *out* indicates the next consumption.
- What if $i == out$?
  - *buffer* is empty, or
  - *buffer* is full.

Interprocess Communication

Producer-Consumer Problem — Bounded Buffer — Shared-Memory Solution

in =			2							
out =	0									
i =	0	1	2	3	4	5	6	7	8	9
buffer	*	*								

- An exercise for you to provide a solution in which *BUFFER_SIZE* items can be in the buffer at the same time.
- *in* indicates the next production, and
- *out* indicates the next consumption.
- What if $i == out$?
  - *buffer* is empty, or
  - *buffer* is full.

Interprocess Communication

Producer-Consumer Problem — Bounded Buffer — Shared-Memory Solution

in =				3						
out =	0									
i =	0	1	2	3	4	5	6	7	8	9
buffer	*	*	*							

- An exercise for you to provide a solution in which *BUFFER_SIZE* items can be in the buffer at the same time.
- *in* indicates the next production, and
- *out* indicates the next consumption.
- What if $i == out$?
  - *buffer* is empty, or
  - *buffer* is full.

Interprocess Communication

Producer-Consumer Problem — Bounded Buffer — Shared-Memory Solution

in =										9
out =	0									
i =	0	1	2	3	4	5	6	7	8	9
buffer	*	*	*	*	*	*	*	*	*	

- An exercise for you to provide a solution in which *BUFFER_SIZE* items can be in the buffer at the same time.
- *in* indicates the next production, and
- *out* indicates the next consumption.
- What if $i == out$?
  - *buffer* is empty, or
  - *buffer* is full.

Interprocess Communication

Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- **Synchronization** will be discussed in great details in Chapter 5.

Interprocess Communication

Message Passing

- 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*)
  - receive(*message*)
- The *message* size is either fixed or variable

Interprocess Communication

Message Passing — Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - Blocking send — the sender is blocked until the message is received
  - Blocking receive — the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - Non-blocking send — the sender sends the message and continue
  - Non-blocking receive — the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous

Interprocess Communication

Producer-Consumer Problem — Message Passing Solution

```
/* producer */
message next_produced;
while (true) {
         /* produce an item in next produced */
        send(next_produced);
}
```

```
/* consumer */
item next_consumed;
while (true) {
        receive(next_consumed);
        /* consume the item in next consumed */
}
```

Interprocess Communication

Message Passing Vs. Shared Memory

- Which is better?
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
- Message passing is easier to implement in a distributed system than shared memory.
- Shared memory can be faster than message passing
  - Message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
  - In shared-memory systems, system calls are required only to establish shared memory regions.
- Message passing provides better performance than shared memory in multi-processing systems
  - Shared memory suffers from cache coherency issues
- As the number of processing cores on systems increases, it is possible that we will see message passing as the preferred mechanism for IPC.

Contents

Communication in Client-Server Systems

Sockets

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port — a number included at start of message packet to differentiate network services on a host
- The socket $161.25.19.8 : 1625$ refers to port 1625 on host $161.25.19.8$
- Communication consists between a pair of sockets
- All ports below 1024 are well known, used for standard services
- Special IP address $127.0.0.1$ (loopback) to refer to system on which process is running

Communication in Client-Server Systems

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- Again uses ports for service differentiation
- Stubs — client-side proxy for the actual procedure on the server
- The client-side stub locates the server and marshalls the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in Microsoft Interface Definition Language (MIDL)

Communication in Client-Server Systems

Pipes

- UNIX pipes are implemented in a similar way, but with the pipe() system call.
  - The output of one process is connected to an in-kernel pipe.
  - The input of another process is connected to that same pipe.

```
prompt>  ls | wc
```

- Ordinary pipes — cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes — can be accessed without a parent-child relationship.

Exercise

Exercise 3.1

- Including the initial parent process, how many processes are created by the following program.

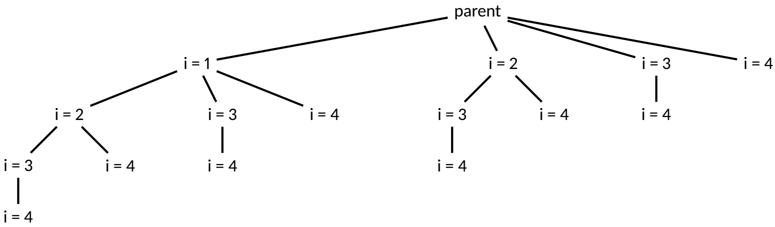
```c
#include <stdio.h>
#include <unistd.h>

int main() {
        int i;
        for (i = 0; i < 4; i ++)
                fork();
        return 0;
}
```

Exercise

Key to Exercise 3.1

```
int i;
for (i = 0; i < 4; i ++)
  fork();
return 0;
```

```
0x1024 ···; some instructions
       implementing fork()
0x2028 incl %eax
0x202c cmpl $0x0004, %eax
0x2030 jne 0x1024
0x2032 retn
```



Exercise

Exercise 3.2

- Draw the diagram of process state.