情報処理 (計算機) システム Ⅱ ・第 13 回

2023 年 7 月 7 日

今回の内容

13.1 プロセス間通信

前回、シェルから

```
y220000@s01612h001:~$ ./myprog <input.data >output.data
```

のように、標準入力や標準出力のリダイレクションを指定してプログラムを起動した場合、シェルや myprog、端末エミュレータなどのプロセスは、各プロセスのファイル記述子を介して、ファイルや他のプロセスと接続され、図 1 のように情報のやり取りを行うことを説明しました。

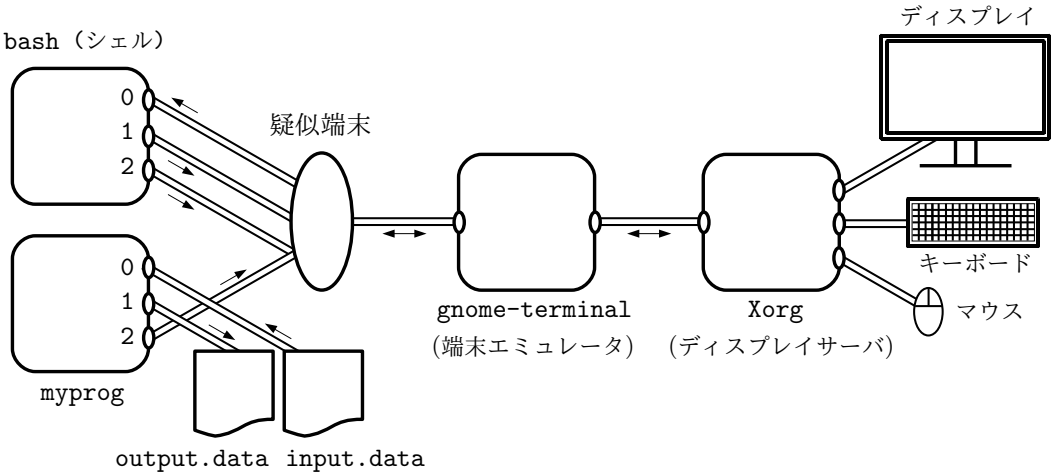

図 1: 標準入力と標準出力がリダイレクトされた myprog のプロセス

このとき、情報のやり取りに使用される通信路はすべて OS (カーネル) が用意したものです。今回は、ユーザプロセス間の通信のために OS が (ユーザプロセスに対して) 提供する仕組みについて説明します。

メモ

13.2 通信の特性の違い

一般に、情報のやり取り (通信) には様々な形態があり、以下のような点で、その特性の違いがあります。

単方向か双方向か　情報が流れる向きが 1 方向のみの通信の場合もあれば、双方向の通信の場合もあります。双方向の場合、2 つの向きの情報の伝達を同時に (並行して) 行うことができる場合と、時間を区切って交互に行わないといけない場合がありますが、前者は**全二重通信** (full duplex)、後者は**半二重通信** (half duplex) と呼ばれます。図 1 における `bash` (シェル) や `myprog` の 3 つのファイル記述子と疑似端末の間では、それぞれ単方向の通信が行われますし、その疑似端末と `gnome-terminal` (端末エミュレータ) の間では双方向の通信が行われます。

バイト単位かパケット単位か　送信側から受信側に伝達される情報が、単なる 1 バイトの情報の並びとして扱われる場合もあれば、**パケット**[1]と呼ばれる (それぞれある大きさを持った) 情報の塊の並びとして扱われる場合もあります。バイト単位で扱われる場合、例えば、送信側が、100 B (バイト) の大きさの情報を一度に送っても、30 B、50 B、20 B の 3 回に分けて送っても、受信側が受け取るデータに違いはなく、全体で 100 B の情報を受け取るだけで、それがどのように区切られていたかは伝わりません。一方、パケット単位の場合、送信側が、ある大きさの情報 (パケット) を送れば、受信側は、その (パケット) の大きさも含めて送られてきた情報を知ることができますので、100 B のパケットが 1 つ送られたのか、それぞれ 30 B、50 B、20 B の大きさの 3 つのパケットが送られたのかを区別することが可能です。

順序性が保証されるかどうか　バイト単位の通信の場合は、通常、送信側が送った情報は送った順に受信側に伝達されることになりますが、パケット単位の通信では、送信側が送信した順序で、受信側がパケットを受け取ることが保証されない場合があります。保証されない場合は、送信側では A、B の順に送ったパケットが、受信側には B、A の順で届くということが起こり得ます。

信頼性があるかないか　バイト単位の通信の場合は、通常、順序性が保証されますので、送信側が送った情報はすべて受信側に伝達されることも保証され、もし、それができない場合には、その障害を検知できるようになっていることが一般的です。このような通信の形態を**信頼性のあるバイトストリーム**と呼びます。

一方、パケット単位の通信の場合、送信側が送ったパケットの受信側への伝達が必ずしも保証されないのが一般的です。しかし、パケットの伝達を保証し、もし、できなかった場合には、それのことが検出できるようにしている場合もあります。

メモ

[1] **メッセージ**と呼んだり、**データグラム**と呼んだりすることもあります。

接続型か接続なしか　通信の相手を特定した上で、まずその相手との専用の通信路を確立し、その後は、その通信路を使って (特に情報の宛先を指定せずに) 情報のやり取りを行う場合もあれば、特に通信路を確立せずに、情報を送る度に宛先を指定する場合もあります。前者で通信路を確立することを**接続**と呼び、接続を行ってから通信を行うものを**接続型**と呼びます。バイト単位の通信 (バイトストリーム) は接続型が一般的です。図 1 に示されている通信は、すべて、まず接続を行って、通信路を確立した上で行われています。一方、パケット単位の通信の場合は、信頼性のある通信の場合を除けば、あまり接続を意識することはありません。信頼性のないパケット単位の通信での「接続」は、送信時に毎回宛先を指定する手順を省略できるという程度の意味しかないのが一般的です。

情報の受け手が単一か複数か　送り手からの情報が、単一の受け手に向かって送信される場合もあれば、複数の受け手に向かって送信される場合もあります。前者の形態の送信を**ユニキャスト**、後者を**マルチキャスト**と呼びます。マルチキャストは信頼性のないパケット単位の通信で行うのが一般的です。

単一の計算機システム内の通信か異なる計算機システム間での通信か　図 1 に示されている通信は、1 つの計算機システム内、つまり、1 つの OS 上のユーザプロセス間での情報のやり取りですが、多くの OS は、このようなプロセス間通信の考え方を拡張し、異なる計算機システム上で動作してるプロセス間の通信ができるような仕組みを提供しています。例えば、Web ブラウザのプロセスは、(通常) 異なる計算機システム上で動作している Web サーバのプロセスと、インターネットを介して通信を行うことで、目的の Web ページを表示します。この場合、通信を行うプロセスがそれぞれ動作している 2 つの計算機システムでは、必ずしも同一の OS が稼働しているとは限りません。そのため、計算機システムの間で通信を行う際の手順に関する約束事を定めておき、たとえ異なる OS 間の通信でも、その定められた手順に従うことで、相互に情報のやり取りができるようにしています。この定められた手順のことを、一般に**通信プロトコル**と呼びます。

メモ

以上のように、プロセス間通信の形態には様々なものがありますが、この中で最も頻繁に利用されるのは、接続ありの信頼性のある (ユニキャストの) バイトストリームで、単方向も双方向もどちらもよく使用されます。図 1 におけるプロセス間の通信はすべてこの形態で行われます。

13.3 Unix 系 OS におけるプロセス間通信のための主な枠組み

Unix 系 OS には、様々なプロセス間通信の枠組みが用意されていますが、その中でも代表的なものとして、パイプとソケットインタフェースという 2 つの枠組みを紹介します。

パイプ　パイプは、Unix 系 OS でプロセス間通信を行うための最も基本的な枠組みで、Unix に古くから存在しているものです。パイプは、信頼性のある単方向[2]のバイトストリームを提供しますが、単一の OS 上で動作しているプロセスの間でしか通信できません。パイプは、`pipe` と呼ばれる次のようなシステムコールによって作成します。

`int pipe(int fds[2])`	パイプを作成し、その通信の両端となるファイル記述子を、`fds[0]` (読み込み用) と `fds[1]` (書き出し用) に格納する。パイプの作成に成功すれば 0 が、失敗すれば -1 が戻り値となる。

`pipe` を呼び出すと、呼び出したプロセスに 2 つの新たなファイル記述子が作成されます。この 2 のファイル記述子は、バイトストリーム通信を行うための端点となり、その一方に `write` したデータを、もう一方から `read` することができます。パイプを作成しただけでは、その通信の端点はどちらも 1 つのプロセスに存在するだけですので、この状態のままではプロセス間の通信に使用することはできません。このため、通常、パイプを生成したプロセスは `fork` し、親と子の両方のプロセスで、同一のパイプの 2 つの端点を共有した状態を作り、その後、それぞれのプロセスで、異なる側の端点を `close` します。これにより、親と子の間で、パイプを介したプロセス間通信を行うことができるようになります。元のプロセスが、パイプを作成した後、2 つの子プロセスを作れば、兄弟のプロセス間での通信路として使えるパイプを作成することもできます。

メモ

ソケット　ソケットインタフェースは、パイプより遅れて Unix のある分派で生まれ、その後 Unix 系 OS の標準規格にも取り込まれた仕組みです。単一の計算機システム上のプロセス間通信にも、異なる計算機システム間のネットワークを介した通信にも使用することができます。パイプが通信路とその端点を同時に作成するのに対して、ソケットインタフェースでは、`socket` システムコールにより、端点をまず作り、その後、端点に、特定の通信プロトコルにおけるアドレス (IP アドレスなど) を付与したり、端点を別の端点と接続して通信路を確立したりといった操作を行い、その後、端点に対してデータの読み書きを行います。`socket` システムコールにより生成された端点は、プロセスのファイル記述子につながり、通常の `write` や `read` のシステムコールでデータの送信や受信をすることができます。

[2] OS によっては双方向の場合もあります。

ソケットインタフェースは、Unix 系 OS でインターネットの通信を行う際の主役となります。以下は、ソケットインタフェースに関連する主なシステムコールです。

`int socket(int domain, int type, int protocol)`	ソケット (通信の端点) を作成し、そのファイル記述子を戻す。`domain` と `protocol` で通信に使用する仕組み (例えば、1 つの計算機システム内のプロセス間通信やインターネット通信など) を指定し、`type` で 13.2 節で解説したような通信の形態 (例えば、接続ありの信頼性のあるバイトストリームなど) を指定する。
`int bind(int fd, struct sockaddr *addr, socklen_t addrlen)`	ファイル記述子 `fd` で指定したソケットに、`addr` と `addrlen` で指定したアドレス (IP アドレスなど) を割り当てる。
`int connect(int fd, struct sockaddr *addr, socklen_t addrlen)`	ファイル記述子 `fd` で指定したソケットを、`addr` と `addrlen` で指定したアドレス (IP アドレスなど) を持つ通信の端点へ接続する。
`int listen(int fd, int backlog)`	ファイル記述子 `fd` で指定したソケットを接続待ち状態にする。接続要求の待ち行列の最大長を `backlog` で指定する。
`int accept(int fd)`	ファイル記述子 `fd` で指定した接続待ち状態のソケットへの次の接続要求を受理し、その接続の通信のための新しいソケットを作成し、そのファイル記述子を戻す。
`int send(int fd, char *buf, size_t len, int flags)`	ファイル記述子 `fd` で指定したソケットの先に接続されている端点へ向かって、`buf` 中のデータを `len` で指定した大きさだけ送信する。この時、`flags` で送り方の詳細を指定する。
`int recv(int fd, char *buf, size_t len, int flags)`	ファイル記述子 `fd` で指定したソケットの先に接続されている端点から送られて来たデータを、最大 `len` で指定した大きさだけ `buf` へ受信する。この時、`flags` で受け方の詳細を指定する。
`int sendto(int fd, char *buf, size_t len, int flags, struct sockaddr *addr, socklen_t addrlen)`	ファイル記述子 `fd` で指定した (接続のない) ソケットから、`addr` と `addrlen` で指定した宛先に向かって、`buf` 中のデータを `len` で指定した大きさだけ送信する。`flags` は送り方の詳細を指定する。
`int recvfrom(int fd, char *buf, size_t len, int flags, struct sockaddr *addr, socklen_t *addrlen)`	ファイル記述子 `fd` で指定したソケットに送られてきたデータを、最大 `len` で指定した大きさだけ `buf` へ受信する。`addr` の指す先には、発信元のアドレスが、最大 `*addrlen` バイトだけ格納され、実際に格納した大きさに `*addrlen` が変更される。`flags` は受け方の詳細を指定する。
`int shutdown(int fd, int how`	ファイル記述子 `fd` で指定したソケットの全二重接続の一方向あるいは双方向を閉じる。どの方向の接続を閉じるかを `how` で指定する。

表中の関数プロトタイプに現れている `struct sockaddr` という型はソケットのアドレスを様々な方法で指定するためのデータ型 (構造体) です。また、`socklen_t` はその長さを表すための型となっています。

Unix 系 OS におけるその他のプロセス間通信のための主な枠組み　Unix 系 OS には他にも次のようなプロセス間通信の枠組みがあります。パイプもソケットもファイル記述子を介してプロセス間通信を行うものでしたが、そうでないものもあります。

名前付きパイプ 名前の付いたパイプを作成することで、`fork` を使わずにプロセス間通信に使用できるようにしたものです。名前付きパイプを介した通信は、通常のパイプと同様にファイル記述子を経由します。

共有メモリ これまで紹介したものは、すべてファイル記述子を介して通信を行うものでしたが、物理メモリのある領域を複数のプロセスで共有しておき、その領域を各プロセスが読み書きすることで通信を行います。共有するメモリ領域は、第 7 回の配布資料で紹介した `mmap` や、Unix のある分派に由来する別のシステムコールを利用して生成できます。

シグナル 何らかの事象を検知したなど、非常に単純な情報をできるだけ早く他のプロセスに通知する仕組みです。シグナルだけではプロセス間でまとまった大きさのデータを受け渡すことはできません。

セマフォ　複数のプロセス間で共有している資源の使用権などを調整するための通信のしくみです。シグナルと同様に、セマフォだけではプロセス間でまとまった大きさのデータを受け渡すことはできません。

情報処理 (計算機) システム Ⅱ ・第 13 回・終わり